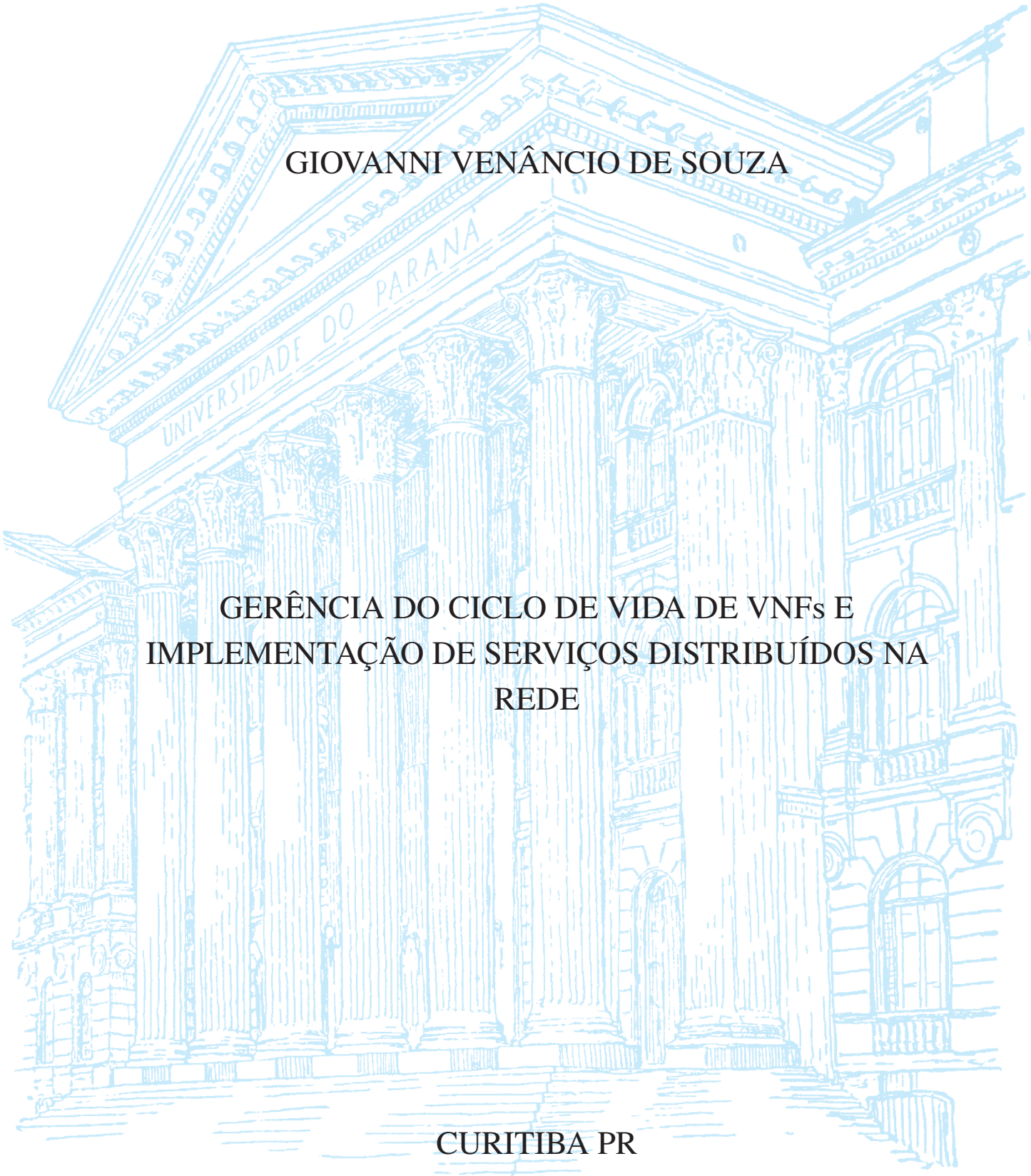
UNIVERSIDADE FEDERAL DO PARANÁ

GIOVANNI VENÂNCIO DE SOUZA



GERÊNCIA DO CICLO DE VIDA DE VNFs E IMPLEMENTAÇÃO DE SERVIÇOS DISTRIBUÍDOS NA REDE

CURITIBA PR
2018

GIOVANNI VENÂNCIO DE SOUZA

GERÊNCIA DO CICLO DE VIDA DE VNFs E IMPLEMENTAÇÃO DE SERVIÇOS DISTRIBUÍDOS NA REDE

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Elias Procópio Duarte Jr..

Coorientador: Rogério Correa Turchetti.

CURITIBA PR
2018

FICHA CATALOGRÁFICA ELABORADA PELO SISTEMA DE BIBLIOTECAS/UFPR
BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

SO729g Souza, Giovanni Venâncio de
Gerência do ciclo de vida de VNFs e implementação de serviços distribuídos na rede / Giovanni Venâncio de Souza. – Curitiba, 2018.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2018.

Orientador: Elias Procópio Duarte Jr.
Coorientador: Rogério Correa Turchetti.

1. Sistemas distribuídos. 2. Virtualização de funções de rede. 3. Difusão confiável. I. Universidade Federal do Paraná. II. Duarte Jr, Elias Procópio. III. Turchetti, Rogério Correa. IV. Título.

CDD: 004

Bibliotecária: Romilda Santos - CRB-9/1214

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **GIOVANNI VENÂNCIO DE SOUZA** intitulada: **Gerência do Ciclo de Vida de VNFs e Implementação de Serviços Distribuídos na Rede**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 29 de Junho de 2018.

ELIAS PROCÓPIO DUARTE JUNIOR
Presidente da Banca Examinadora

CARLOS RANIERY PAULA DOS SANTOS
Avaliador Externo

LUIS CARLOS ERPEN DE BONA
Avaliador Interno

ALDELIR FERNANDO LUIZ
Avaliador Externo

ROGÉRIO CORRÊA TURCHETTI
Avaliador Externo

Dedico este trabalho aos meus pais Celso e Silvana e à minha namorada Natissa.

Agradecimentos

Em primeiro lugar, agradeço o meu orientador Prof. Elias. Foi realmente ótimo trabalhar com você nesses últimos anos. Foram muitas reuniões, trabalhos e comemorações! Obrigado pelo apoio e por sempre confiar no meu trabalho.

Ao meu coorientador, Prof. Rogério Turchetti, foi um prazer imenso trabalhar com você. Sempre disposto a ajudar, posso garantir que a sua coorientação foi essencial no meu trabalho.

Aos meus pais Celso e Silvana, que sempre me apoiaram e incentivaram, muito obrigado!

À minha namorada Natissa, agradeço por estar comigo durante todos estes anos, sempre me acalmando nos momentos de dificuldade. Sei que posso contar com você para tudo.

Aos meus amigos André e Lucas, pelos anos de amizade.

Agradeço também a todas as pessoas que de alguma forma contribuíram para a realização deste trabalho.

Resumo

A Virtualização de Funções de Rede (*Network Function Virtualization* - NFV) oferece uma alternativa para projetar, desenvolver e gerenciar serviços de rede. Através de técnicas de virtualização, serviços que antes eram oferecidos em hardware especializado agora são disponibilizados como Funções Virtualizadas de Rede (*Virtualized Network Functions* - VNFs) e executam em hardware genérico (*e.g.*, arquitetura x86). Esta dissertação tem como objetivo investigar as vantagens que NFV oferece para implantar e gerenciar VNFs dentro da própria rede. As contribuições são divididas em duas partes. A primeira parte explora o gerenciamento de VNFs e a segunda descreve implementações de VNFs. As soluções existentes para o gerenciamento do ciclo de vida de VNFs são complexas e exigem um grande entendimento da infraestrutura subjacente. Uma das contribuições deste trabalho é a especificação de uma arquitetura para o VNF *Manager* (VNFM), de forma a permitir a compatibilidade entre diferentes plataformas NFV, além de simplificar as operações de gerência, na medida em que diminui a necessidade de que o operador das funções conheça os detalhes da infraestrutura virtualizada. Resultados experimentais demonstram a efetividade da arquitetura proposta, avaliando individualmente cada operação do ciclo de vida. A segunda contribuição aborda a sincronização consistente do plano de controle distribuído em Redes Definidas por Software (*Software Defined Network* - SDN). Para tanto, é proposta a *VNF-Consensus*, uma VNF que implementa o algoritmo Paxos para garantir a consistência entre múltiplos controladores SDN. Com a *VNF-Consensus*, os controladores são desacoplados das tarefas de sincronização e podem executar em paralelo suas atividades no plano de controle. Resultados experimentais demonstram os benefícios obtidos pelo uso da *VNF-Consensus*, em especial na redução na carga dos controladores. Por fim, a última contribuição detalha o *AnyBone*: um *backbone* virtual que implementa uma VNF para oferecer serviços de difusão de mensagens implementados na própria rede. Os diversos serviços de difusão são frequentemente utilizados para a construção de aplicações distribuídas e tolerante a falhas. Em geral, a difusão é implementada na própria aplicação, aumentando a complexidade no desenvolvimento. Neste contexto, o *AnyBone* oferece uma primitiva de difusão confiável e três primitivas de difusão ordenada de mensagens: FIFO (*First-In First-Out*), causal e atômica. Em particular, a solução proposta garante a ordem das mensagens através da utilização um sequenciador, denominado de *VNF-Sequencer*, que é implementado como uma VNF. Resultados experimentais são apresentados e a *VNF-Sequencer* é avaliada em termos de vazão e latência.

Palavras-chave: Sistemas Distribuídos, Virtualização de Funções de Rede, Difusão Confiável.

Abstract

Network Function Virtualization (NFV) allows the deployment of network services that have been usually offered as middleboxes based on specialized hardware as Virtualized Network Functions (VNFs) that are executed on off-the-shelf hardware. The contributions of this dissertation are of two types: the first is related to the investigation of strategies to improve NFV management practices, and the second is related to the investigation of the feasibility of offering distributed systems services within the network. Current solutions for VNF lifecycle management are complex and require a deep understanding of the underlying infrastructure. One of the contributions of this work is the specification of an architecture for the ETSI VNF Manager (VNFM). The proposed VNFM improves the compatibility of different NFV platforms, and simplify management operations and reduce the level of knowledge required about details of the infrastructure. Experimental results demonstrate the effectiveness of the proposed management architecture, each lifecycle operation was individually evaluated. The second contribution of this dissertation by itself consists of two parts. The first was proposed in the context of the requirements for consistency whenever the control plane of Software Defined Networks (SDN) is distributed - usually it is centralized, but there are several reasons to employ multiple controllers. We propose *VNF-Consensus*, a VNF that implements the Paxos algorithm to ensure consistency among multiple SDN controllers. With *VNF-Consensus*, controllers are decoupled from synchronization tasks and can execute their tasks without this extra burden. Experimental results demonstrate the benefits obtained by using *VNF-Consensus*, especially the reduction of the load on controllers. Finally, the last contribution introduces *AnyBone*: a virtual backbone that implements a VNF to provide broadcast services that are deployed on the network itself. Several broadcast services which are often used for building distributed and fault tolerant applications were implemented. In general, these broadcast abstractions are implemented as applications at end-user hosts. In this context, *AnyBone* provides – within the network – a reliable broadcast primitive and three ordered broadcast primitives: FIFO (First-In First-Out), causal, and atomic. In particular, the proposed solution ensures the order of the messages by using a sequencer, called *VNF-Sequencer*, which was implemented as a VNF. Experimental results are presented and the *VNF-Sequencer* is evaluated in terms of throughput and latency.

Keywords: Fault Tolerance, Network Function Virtualization, Reliable Broadcast.

Lista de Figuras

Lista de Acrônimos

API	*Application Programming Interface*
BB	*Broadcast-Broadcast*
BSS	*Business Support Systems*
CAPEX	*CAPital EXpenditures*
CAS	*Compare-and-Set*
EMS	*Element Management*
ETSI	*European Telecommunications Standards Institute*
FPC	*Fast Paxos-based Consensus*
FCAPS	*Fault, Configuration, Accounting, Performance, Security, Management*
FIFO	*First-In First-Out*
ISG	*Industry Specification Group*
JSON	*JavaScript Object Notation*
NFV	*Network Function Virtualization*
NFVI	*NFV Infrastructure*
NFV-MANO	*NFV MANagement and Orchestration*
ODL	*OpenDaylight*
OPEX	*OPerational EXpenditures*
OSM	*Open Source MANO*
OSS	*Operations Support Systems*
REST	*REpresentational State Transfer*
SDN	*Software Defined Network*
SFC	*Service Function Chaining*
TOSCA	*Topology and Orchestration Specification for Cloud Applications*
UB	*Unicast-Broadcast*
UUB	*Unicast-Unicast-Broadcast*
VIM	*Virtualized Infrastructure Manager*
VM	*Virtual Machine*
VNF	*Virtualized Network Function*
VNFD	*VNF Descriptor*
VNFM	*VNF Manager*
YAML	*YAML Ain't Markup Language*

Sumário

1 Introdução

As redes de computadores são constituídas de diversos dispositivos disponibilizados tanto em hardware como em software. Alguns destes dispositivos, denominados de *middleboxes*, fornecem diversos serviços como filtragem de pacotes, balanceamento de carga, detecção de intrusão, entre outros [Sekar et al. 2012]. Apesar de oferecer funcionalidades indispensáveis para uma infraestrutura eficiente e segura, estes serviços representam uma importante fração das despesas de capital (*CAPital EXpenditures* - CAPEX) e despesas operacionais (*OPerational EXpenditures* - OPEX) [Cotroneo et al. 2014] de uma rede, já que cada um desses componentes requer hardware especializado. Em especial, grande parte do OPEX/CAPEX é referente à manutenção, instalação, configuração e aquisição de hardware, uma vez que é necessária a substituição dos dispositivos quando estes tornam-se desatualizados. Além disso, muitos *middleboxes* são complexos para implantar, gerenciar e solucionar problemas [Sherry et al. 2012].

A Virtualização de Funções de Rede (*Network Function Virtualization* - NFV) surge como uma alternativa para projetar e gerenciar serviços de rede, aplicando tecnologias de virtualização para implementar hardware especializado (*i.e.*, *middleboxes*) em hardware de prateleira [Chiosi et al. 2012]. Dessa forma, os serviços de rede são disponibilizados por meio de dispositivos virtuais, denominados de Funções Virtualizadas de Rede (*Virtualized Network Functions* - VNFs). Visto que as funcionalidades dos dispositivos de rede estão executando de maneira virtualizada, a inclusão de novos serviços, bem como a alteração e atualização de serviços já existentes, não necessitam da instalação de equipamentos de hardware especializados, diminuindo custos e flexibilizando o provisionamento de serviços [Mijumbi et al. 2016].

Em comparação com os dispositivos de hardware, as VNFs reduzem o OPEX e CAPEX [Cotroneo et al. 2014], além de permitirem economia de energia e de espaço físico [Han et al. 2015]. Além disso, a flexibilidade no gerenciamento dos serviços da rede é melhorada na medida que as VNFs são facilmente instanciadas, utilizadas e removidas. Outra vantagem significativa das VNFs está no ajuste automático dos recursos disponibilizados para cada função virtualizada, já que é possível aumentar ou diminuir o uso de recursos computacionais de acordo com a demanda necessária, fazendo uso eficiente dos recursos disponíveis.

Existem diversos desafios para a utilização efetiva da NFV, incluindo o gerenciamento e a garantia de desempenho das VNFs. Neste sentido, o grupo ISG (*Industry Specification Group*) criado pela ETSI (*European Telecommunications Standards Institute*) vem propondo especificações para a padronização de uma arquitetura NFV [ETSI 2016]. O objetivo principal desta arquitetura é facilitar a gerência dos serviços virtualizados, além de permitir o suporte à execução de VNFs de diferentes desenvolvedores.

Tendo em vista os conceitos expostos, esta dissertação tem como objetivo explorar as vantagens que a tecnologia de NFV oferece para implementar e gerenciar VNFs dentro da própria rede. Este trabalho apresenta um conjunto de contribuições que é classificado em duas partes: *(i)* gerenciamento de VNFs; *(ii)* implementação e execução de serviços distribuídos na rede. Na primeira parte é tratado o gerenciamento de VNFs e são investigadas as plataformas existentes que implementam a arquitetura proposta pela ETSI. Em particular, é analisada a

complexidade do uso destas plataformas, bem como lacunas nas suas funcionalidades. Já na segunda parte, são descritas duas contribuições que tratam especificamente da implementação de funções virtualizadas de rede. A primeira descreve uma proposta de uma VNF para sincronização consistente de uma Rede Definida por Software (*Software Defined Network* - SDN) composta por múltiplos controladores. Por fim, é apresentada uma VNF que oferece múltiplos serviços de difusão confiável e ordenada de mensagens dentro da própria rede. É importante ressaltar que, apesar da NFV ser originalmente proposta no contexto onde dispositivos de hardware são virtualizados e executados na rede, este trabalho investiga uma abordagem ligeiramente diferente, onde os serviços que serão virtualizados são aqueles que são geralmente executados na camada de aplicação.

Primeiramente, este trabalho apresenta a proposta de uma arquitetura de um VNF *Manager* (VNFM). O VNFM é um dos principais módulos da arquitetura NFV proposta pela ETSI, responsável principalmente pela gerência do ciclo de vida das VNFs. O ciclo de vida consiste basicamente da instanciação, atualização e remoção das funções virtualizadas, podendo compreender também outras ações, como o ajuste automático de recursos.

Atualmente existem diversas plataformas que implementam o VNFM proposto pela ETSI [Tacker 2017, OpenBaton 2017, ETSI 2017, ONAP 2017, Rift.io 2017]. Em geral, estas plataformas exigem uma grande quantidade de procedimentos para realizar a gerência do ciclo de vida das VNFs. Consequentemente, tais plataformas tornam-se complexas, demandando do operador das funções um grande entendimento da infraestrutura do sistema [Shen et al. 2015]. Além disso, estas soluções são pouco flexíveis, dificultando a integração entre diferentes tecnologias NFV. Por fim, a maioria destas plataformas não gerenciam as VNFs de maneira completa, tornando-se necessária a utilização de ferramentas adicionais para suprir as lacunas destas plataformas. Em específico, as soluções atuais fornecem as funcionalidades para gerenciar a máquina que irá hospedar a função virtualizada (*i.e.*, gerência do hardware), porém sem oferecer meios para gerenciar a função propriamente dita (*i.e.*, gerência do software).

A arquitetura proposta neste trabalho tem como objetivo suprir as deficiências descritas acima. Para simplificar as operações de gerência, é definida uma API de alto nível que permite o controle do ciclo de vida das VNFs através de requisições feitas pelo usuário. Essa API abstrai e automatiza diversos procedimentos que antes eram executados pelo operador da função. Além disso, a arquitetura proposta possui outras duas APIs que permitem a compatibilidade entre diferentes tecnologias NFV. Por fim, é proposto um submódulo do VNFM que utiliza todas as APIs definidas para gerenciar as VNFs de maneira completa, desde o nível de hardware até o nível de software. Um protótipo, denominado de *vCommander* implementa a arquitetura proposta. Resultados experimentais analisam o desempenho individual de cada operação e demonstram a efetividade do *vCommander*.

O principal objetivo de redes SDN é separar o plano de controle do plano de dados. O controle é, em geral, centralizado e responsável por tomar decisões que ocorrem no plano de dados. Nesta abordagem centralizada, o estado da rede é determinado por um controlador [Ho et al. 2016]. Entretanto, manter o plano de controle centralizado trás desafios em termos de disponibilidade, escalabilidade e desempenho [Canini et al. 2015]. Neste sentido, há um consenso de que o plano de controle precisa ser distribuído, o que na prática não é uma tarefa trivial [Schiff et al. 2016]. Um dos maiores desafios em SDN é garantir a consistência das operações realizadas na rede que precisam ser consistentemente sincronizadas entre os múltiplos controladores.

Neste trabalho é proposta a *VNF-Consensus*, uma função virtualizada de rede responsável por garantir a consistência de operações entre múltiplos controladores SDN em um plano de controle distribuído. O objetivo desta abordagem é desacoplar os controladores das tarefas de

sincronização, visto que tais tarefas são computacionalmente custosas, evitando então o aumento na carga de trabalho dos controladores. Cada controlador possui acesso a uma instância da *VNF-Consensus* através da qual poderá enviar ações para serem sincronizadas. Resultados experimentais demonstram os benefícios da utilização da *VNF-Consensus*. Em particular, sincronizar o plano de controle sem aumentar a carga de trabalho dos controladores implica em ganhos significativos na vazão e latência das operações.

Por fim, a última contribuição aborda a implementação de difusão confiável e ordenada. A difusão confiável é uma abstração importante para o desenvolvimento de aplicações distribuídas e tolerante a falhas, e garante que as mensagens enviadas para um conjunto de processos seja entregue por todos os processos corretos do sistema [Défago et al. 2004]. Entretanto, a difusão confiável por si só não garante a ordem em que estes processos irão entregar as mensagens. Neste caso, é necessário utilizar algoritmos de difusão confiável e ordenada, que garantem a ordem em que as mensagens serão entregues, tais como a ordem FIFO (*First-In First-Out*), causal [Lamport 1978] e atômica.

Na prática, a implementação destas difusões não é uma tarefa trivial. Em geral, as difusões são implementadas na própria aplicação, o que por consequência aumenta significativamente a complexidade para o seu desenvolvimento [Li et al. 2016]. Neste sentido, este trabalho propõe o *AnyBone*: um *backbone* virtual baseado em NFV que oferece as primitivas de difusão para garantir a entrega confiável e ordenada das mensagens transmitidas na rede. Dessa forma, a própria rede oferece às aplicações meios para garantir a entrega e ordem das mensagens, diminuindo a complexidade no desenvolvimento das aplicações distribuídas.

O *AnyBone* define a ordem das mensagens através do uso de um sequenciador que também está executando dentro da própria rede. Este sequenciador, denominado de *VNF-Sequencer*, é implementado e disponibilizado como uma VNF, sendo responsável por intermediar toda a troca de mensagens entre as aplicações do sistema, de forma a garantir todas as propriedades de comunicação. Assim, o *AnyBone* oferece os seguintes tipos de difusão: difusão confiável, difusão atômica, difusão atômica FIFO e difusão atômica causal. O *AnyBone* foi implementado e os resultados experimentais avaliam a latência e a vazão da *VNF-Sequencer*.

O restante deste trabalho está organizado da seguinte forma. Os Capítulos 2 e 3 apresentam conceitos básicos de sistemas distribuídos e de virtualização de funções de rede que serão utilizados no decorrer deste trabalho. O Capítulo 4 apresenta uma especificação completa para o gerenciamento de VNFs. Em sequência, o Capítulo 5 descreve a *VNF-Consensus* e o Capítulo 6 detalha o *AnyBone*. Por fim, as conclusões são apresentadas no Capítulo 7.

2 Sistemas Distribuídos: Conceitos Básicos

Este capítulo tem como objetivo descrever brevemente os conceitos básicos de sistemas distribuídos utilizados no presente trabalho. A Seção 2.1 apresenta as definições elementares de sistemas distribuídos. A Seção 2.2 descreve os detectores de falhas, sua classificação e suas respectivas propriedades. Na Seção 2.3 é feita uma introdução sobre o consenso. Por fim, a Seção 2.4 descreve os principais algoritmos de difusão confiável e ordenada.

2.1 Definição de Sistemas Distribuídos

Um sistema distribuído é um conjunto de processos (*i.e.*, programas em execução), executando em máquinas distintas, cooperando para resolver uma determinada tarefa em uma rede, através da troca de mensagens [Kshemkalyani and Singhal 2011]. Os problemas resolvidos em um sistema distribuído podem variar desde uma aplicação específica, por exemplo aplicações *web*, banco de dados distribuídos e sistemas operacionais distribuídos, até aplicações de propósito geral (*e.g.*, MPI–*Message Passing Interface*). Existem diversos motivos para utilizar um sistema distribuído em vez de um sistema centralizado, como por exemplo o compartilhamento de recursos, escalabilidade, entre outros.

Uma aplicação deve executar as suas tarefas de acordo com a sua especificação. No entanto, falhas podem gerar comportamentos inconsistentes, possivelmente gerando resultados incorretos. Entretanto, é essencial garantir que o sistema continue operacional, mesmo com a falha de alguns de seus componentes. Por esse motivo, os sistemas devem satisfazer a propriedade de tolerância a falhas, garantindo em qualquer circunstância a execução correta do sistema [Kshemkalyani and Singhal 2011, Avizienis et al. 2004].

Para um sistema garantir a tolerância a falhas, é necessário satisfazer os seguintes requisitos [Avizienis et al. 2004]:

- Disponibilidade (*availability*): capacidade de oferecer retomada de serviço em casos de interrupção.
- Confiabilidade (*reliability*): capacidade de oferecer serviço correto e contínuo (*i.e.,* sem interrupções).
- Segurança (*safety*): capacidade de evitar consequências desastrosas para o usuário.
- Integridade (*integrity*): capacidade de evitar a alteração imprópria do sistema.
- Manutenção (*Maintainability*): capacidade de sofrer alterações e modificações.

Como descrito acima, sistemas distribuídos devem ser capazes de permanecer operacionais mesmo após a falha de seus componentes. Uma *failure* (falha no serviço) pode ser identificada através de um resultado incorreto para o qual foi especificado. *Failures* são provocadas por um ou mais erros. Erros são manifestações de *faults* (falhas em um ou mais componentes do sistema). Existem diversos cenários onde processos podem falhar. Sendo assim, foram propostos vários modelos de falha que especificam o modo pelo qual os componentes falham. Modelos de falha são particularmente importantes pois o algoritmo utilizado para resolver uma determinada tarefa varia de acordo com o modelo adotado [Kshemkalyani and Singhal 2011]. Os principais modelos de falha são descritos a seguir.

O modelo de falhas por parada (*crash*) especifica a parada completa do componente, não executando nenhuma computação local, nem o envio de mensagens para outros processos. Portanto, os outros processos corretos não possuem a informação de que este processo está falho. É importante ressaltar que após falhar, o componente nunca se recupera. Por sua vez, a falha por omissão especifica que o componente envia e/ou recebe apenas parte das mensagens que deveria produzir. Por exemplo, essa falha é causada por erros de *buffer overflow* ou congestionamento da rede [Cachin et al. 2011]. Na falha bizantina, os processos possuem um comportamento arbitrário, produzindo mensagens diferentes da especificação. A causa desse comportamento pode ser oriunda de erros na implementação do sistema até ações maliciosas. Note que a falha bizantina por ser mais abrangente em sua definição, engloba as falhas por omissão e por parada.

Existem ainda variações dos modelos descritos acima. Por exemplo, é possível considerar a recuperação de um processo após a sua falha. Nesse modelo, denominado de *crash-recovery*, os processos mantém salvo o seu estado interno, sendo possível a restauração de suas informações após a sua recuperação. Ainda, o modelo *fail-stop* considera que os processos falham por parada. Porém, ao contrário do modelo *crash*, o modelo *fail-stop* garante que os outros processos corretos saibam da falha deste processo. Por fim, a falha de temporização ocorre quando um processo não respeita os limites de tempo para o envio de mensagens. Esse tipo de falha ocorre apenas em sistemas que adotam o modelo síncrono (os modelos temporais são detalhados na Seção 2.2), uma vez que no modelo assíncrono não assume nenhuma premissa temporal.

2.2 Detectores de Falhas

Um aspecto importante a considerar em sistemas distribuídos é o comportamento dos processos com relação ao tempo. Isto é, a questão *"é possível determinar limites de tempo de resposta de um processo em um sistema?"* deve ser analisada ao construir uma aplicação que irá executar em um sistema distribuído. Sendo assim, podemos assumir vários modelos temporais. Entre eles o modelo síncrono, assíncrono e os diversos modelos parcialmente síncronos [Mullender et al. 1993, Cachin et al. 2011]. No modelo síncrono, existe um limite superior conhecido em que um processo irá executar uma determinada tarefa. É importante notar que esta tarefa pode envolver o recebimento e/ou a transmissão de mensagens bem como o processamento local dos dados. Na prática, uma possível maneira de se obter um modelo síncrono é através da utilização de *timeouts*–parâmetros que estabelecem um tempo máximo de resposta para uma mensagem e que são utilizados para o controle do tempo. Por outro lado, modelos assíncronos não fazem nenhuma consideração sobre esses limites temporais. Em outras palavras, esse modelo é caracterizado pela ausência de conhecimento temporal de qualquer ordem. Por fim, existem os modelos parcialmente síncronos. Estes modelos definem níveis intermediários de sincronismo (*i.e.*, entre o síncrono e o assíncrono), e por isto são chamados de parcialmente síncronos.

Uma vez caracterizado o modelo temporal de sistemas distribuídos, é possível observar que o modelo síncrono é mais restrito e não reflete a realidade da maioria dos ambientes reais, uma vez que não é uma tarefa trivial conhecer limites de tempo. Apesar de modelos assíncronos serem mais fáceis de implantar em ambientes reais, não é possível fazer nenhuma afirmação com relação ao tempo de resposta dos processos neste sistema. Isso implica, por exemplo, em não poder distinguir um processo falho de um processo lento. Sem essa distinção é impossível a execução de algoritmos determinísticos, tal como o algoritmo de consenso (descrito na Seção 2.3). Essa impossibilidade é conhecida como impossibilidade FLP, descrita em [Fischer et al. 1985].

Inspirado nesse impasse, a abstração de detectores de falhas é proposta em [Chandra and Toueg 1996]. Nessa abstração, cada processo tem acesso a um módulo local que possui informações sobre quais processos estão falhos e quais estão corretos. Essas informações são obtidas através do monitoramento aos processos do sistema. São utilizados *timeouts* para determinar se um processo é classificado como falho ou correto.

Por outro lado, as informações obtidas pelos detectores de falhas podem ser incorretas. Por exemplo, um processo falho pode ser considerado correto. Esse tipo de erro pode acontecer, por exemplo, se um processo falha logo após o detector coletar o estado atual do processo. Outra informação imprecisa que um detector pode obter é considerar um processo correto como falho. Isso pode acontecer quando um processo está correto, porém está mais lento que os demais, atrasando a resposta por um tempo indeterminado. Além disso, como não é garantido nenhum limite de tempo nas respostas dos processos, detectores de falhas de processos diferentes podem ter visões diferentes. Por exemplo, o detector FD_i (*Failure Detector* hospedado no processo i) pode considerar o processo p_x como falho ao passo que o detector FD_j considere o mesmo processo p_x como correto. Assim, os detectores de falhas são definidos como não confiáveis [Chandra and Toueg 1996].

Em [Chandra and Toueg 1996], os detectores de falhas são caracterizados por duas propriedades: completude (*completeness*) e precisão (*accuracy*). A completude garante que todos os processos que estão em um estado falho são suspeitados. Por outro lado, a precisão garante que nenhum processo correto é suspeitado. Note que estas propriedades são complementares, uma vez que um detector de falhas pode suspeitar de todos os processos (satisfazendo a completude) ou não fazer nenhuma suspeita (satisfazendo a precisão). Existem dois tipos de completude e quatro tipos de precisão e são descritos a seguir:

- Completude Forte: em algum momento, todos os processos falhos são suspeitados por todos os processos corretos.
- Completude Fraca: em algum momento, todos os processos falhos são suspeitados por algum processo correto.
- Precisão Forte: processos corretos nunca são suspeitados por processos corretos.
- Precisão Fraca: algum processo correto nunca é suspeitado por nenhum processo correto.
- Precisão Eventual Forte: em algum momento, nenhum processo correto é suspeitado por outro processo correto.
- Precisão Eventual Fraca: em algum momento, algum processo correto não é suspeitado por nenhum processo correto.

Como as propriedades de completude e precisão devem ser complementares, através da combinação das propriedades é possível deduzir oito classes de detectores de falhas, apresentadas

Tabela 2.1: Classes dos detectores de falhas.

	Precisão			
Completude	Forte	Fraca	Eventual Forte	Eventual Fraca
Forte	Perfeito $\mathcal{P}$	Forte $\mathcal{S}$	Eventualmente Perfeito $\Diamond\mathcal{P}$	Eventualmente Forte $\Diamond\mathcal{S}$
Fraca	$\mathcal{L}$	Fraca $\mathcal{W}$	$\Diamond\mathcal{L}$	Eventualmente Fraca $\Diamond\mathcal{W}$

na Tabela 2.1. Cada classe é definida pela combinação de uma propriedade de completude com uma propriedade de precisão. Por exemplo, o detector $\mathcal{P}$ denominado de Perfeito satisfaz as propriedades de Completude Forte e Precisão Forte e é a classe mais forte de todas. Por outro lado, a classe mais fraca e menos restrita é a $\Diamond\mathcal{W}$, que deve satisfazer as propriedades de Completude Fraca e Precisão Eventual Fraca. Essa classe de detector de falhas é importante, uma vez que ela é suficiente para permitir o consenso [Chandra et al. 1996]. Isso acontece devido a equivalência entre as propriedades Completude Fraca e Completude Forte, já que a transformação pode ser facilmente realizada através da transmissão de uma suspeita para todos os outros detectores.

Existem diversos modelos para a implementação de detectores de falhas. Em geral, o monitoramento ocorre através de mensagens de monitoramento realizada por trocas de mensagens considerando *timeouts*. Além disso, cada processo possui acesso a uma instância local do detector de falhas [Chandra and Toueg 1996, Cachin et al. 2011]. Em [Felber et al. 1999], os autores dividem os modelos de monitoramento em duas categorias: modelo *push* e modelo *pull*. No modelo *push*, os processos que estão sendo monitorados enviam mensagens do tipo *heartbeat* para o detector. Se após um período de tempo (determinado pelo parâmetro de *timeout*) o detector não recebe essa mensagem, o estado do respectivo processo é alterado de correto para falho. Por outro lado, no modelo *pull*, o detector de falhas envia mensagens do tipo *Are you alive?* para os processos que está monitorando. Se um processo não responde dentro do intervalo de tempo definido pelo parâmetro de *timeout*, este processo é considerado falho. Note que este segundo modelo gera um número maior de mensagens na rede.

2.3 Consenso

Processos de um sistema distribuído frequentemente precisam entrar em acordo em relação às ações executadas dentro do sistema. Por exemplo, aplicações distribuídas frequentemente utilizam replicação de dados para garantir a disponibilidade do sistema e mascarar falhas nos servidores [Li et al. 2016]. O acordo é necessário nas mais diversas aplicações, por exemplo em replicação de banco de dados, difusão atômica de mensagens [Hadzilacos and Toueg 1994], entre outras.

Uma maneira de realizar a decisão do acordo entre todos os processos de um sistema distribuído é através da utilização de um algoritmo de consenso. Esse tipo de algoritmo garante que todos os participantes irão entrar em acordo sobre um único valor dentro de um conjunto de valores, possivelmente diferentes, propostos inicialmente pelos processos participantes [Borran et al. 2012]. Além disso, o consenso deve funcionar mesmo com a falha de alguns de seus participantes.

O consenso pode ser especificado utilizando duas primitivas: *propose* e *decide*. Inicialmente, existem valores iniciais que são propostos através da execução do *propose*. Uma vez que

o consenso é atingido, todos os processos corretos devem optar pelo mesmo valor, através da execução do *decide*. O algoritmo de consenso deve satisfazer as quatro propriedades descritas a seguir [Cachin et al. 2011]:

- Terminação: todo processo correto em algum momento decide algum valor.
- Validade: se um processo decide um valor v, então v foi proposto por algum processo correto.
- Integridade: nenhum processo decide mais de uma vez em uma mesma execução do algoritmo.
- Acordo: dois processos corretos não decidem por valores diferentes.

O Paxos é um algoritmo de consenso tolerante a falhas projetado para replicação de máquina de estado [Lamport 1998, Lamport 2001]. O algoritmo assume o modelo de falhas de parada com recuperação (*crash-recovery*). No Paxos, os processos podem assumir os seguintes papéis: *proposers*, *acceptors* e *learners*. Os *proposers* propõem um valor, os *acceptors* escolhem um valor e os *learners* aprendem o valor decidido. Um único processo pode assumir qualquer um destes papéis ou múltiplos papéis simultaneamente. O algoritmo Paxos é ótimo em termos de resiliência [Lamport 2006]: para tolerar f falhas ele requer $2f + 1$ *acceptors*.

Para garantir que diversas execuções do consenso sejam realizadas, o algoritmo executa instâncias separadas do Paxos [Lamport 2001]. Cada instância corresponde a uma execução do consenso e está associada a um valor decidido. Uma instância do Paxos executa tipicamente em duas fases, descritas a seguir a partir de um cenário exemplo.

A Figura 2.1 apresenta um cenário de execução do Paxos em duas fases, com dois *proposers* (P_0 e P_1), três *acceptors* (A_0 , A_1 e A_2) e dois *learners* (L_0 e L_1). O *proposer* P_1 inicia o consenso ao receber um valor v através de uma requisição *propose*. Durante a primeira fase, P_1 seleciona um número único de rodada *n-rod* e o envia em uma solicitação *prepare* aos *acceptors*. Ao receber uma solicitação $prepare$ com um número de rodada maior que qualquer rodada que o $acceptor$ tenha recebido previamente, o $acceptor$ responde ao $proposer$ prometendo que irá rejeitar qualquer requisição futura com números de rodada menores. Se o $acceptor$ já aceitou um valor para a instância atual (explicado a seguir), ele irá retornar esse valor ao *proposer* juntamente com o número de rodada recebido quando o valor foi aceito. Na Figura 2.1, a resposta dos $acceptors$ é realizada em uma requisição $prepareResp$ com as seguintes informações: *n-rod* é o número de rodada, *v-rod* é a rodada em que o comando foi aceito e *v-accept* é o valor aceito. Caso não haja valor aceito, os dois últimos parâmetros são nulos. Quando o $proposer$ recebe respostas de um quórum, ou seja, uma maioria de $acceptors$, o algoritmo segue para a segunda fase.

Na segunda fase, o *proposer* seleciona um valor de acordo com a seguinte regra: se nenhum *acceptor* no quórum de respostas aceitou um valor, o *proposer* pode selecionar um novo valor para a instância (no caso, o valor v recebido na requisição *propose*). No entanto, se qualquer um dos *acceptors* retornou um valor na primeira fase, o *proposer* escolhe o valor com o maior número de rodada. O *proposer* então envia uma solicitação *accept* com o número de rodada usado na primeira fase e o valor escolhido (*v-selected*) para os *acceptors*. Quando recebem essa solicitação, os *acceptors* enviam mensagens de confirmação (*acceptResp*) ao *proposer* e aos *learners*. Quando um quórum de *acceptors* aceita o valor, o consenso é atingido.

Se múltiplos *proposers* simultaneamente executam o procedimento anterior para a mesma instância, então nenhum *proposer* pode conseguir executar as duas fases do protocolo e alcançar o consenso. Por exemplo, P_0 e P_1 na Figura 2.1 podem ficar competindo indefinidamente

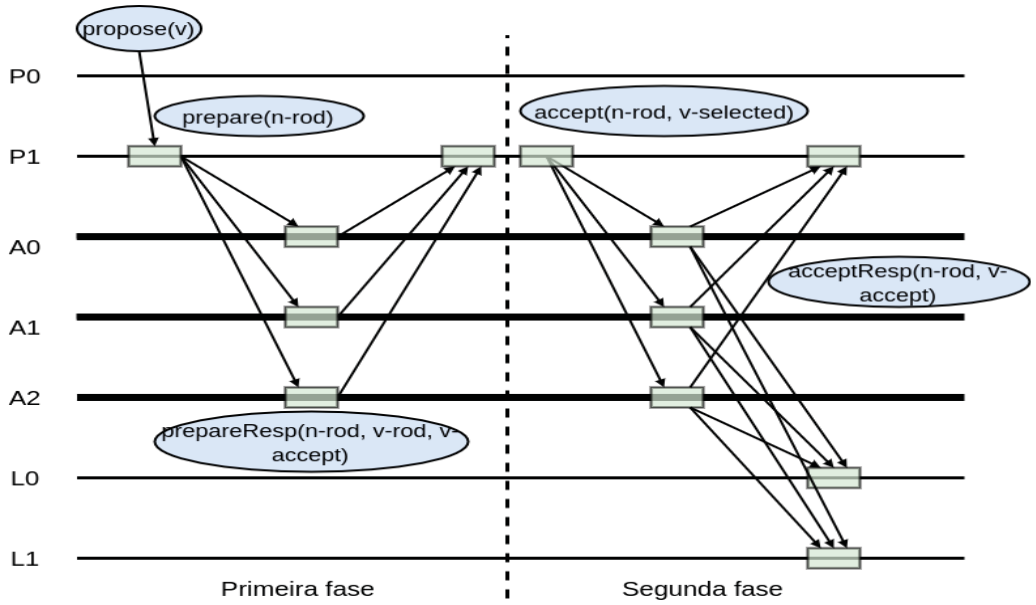


Figura 2.1: Paxos em duas fases (fonte [de Camargo and Duarte 2017]).

pelo maior número de rodada. Para evitar esse cenário, um processo coordenador pode ser escolhido. Neste caso, *proposers* submetem valores ao coordenador, que então executa a primeira e a segunda fases do protocolo. Se o coordenador falha, outro processo assume a sua função. O algoritmo Paxos garante a consistência mesmo com múltiplos *proposers* concorrentes e garante a terminação na presença de um único coordenador.

2.4 Difusão Confiável

Em um sistema distribuído é comum a necessidade dos processos enviarem mensagens para todos os outros processos da rede. O envio de uma mensagem de um processo para todos os outros processos de um sistema distribuído é denominado de difusão (*broadcast*).

Se um processo deseja transmitir uma mensagem de forma confiável, *i.e.*, de forma a garantir que todos os processos corretos na rede entreguem a mensagem, então deve-se utilizar a difusão confiável (*reliable broadcast*) [Défago et al. 2004]. A difusão confiável é necessária com frequência, podendo ser considerada essencial para a construção de aplicações tolerantes a falhas [Pedone and Schiper 2003].

A difusão é formalmente definida através de duas primitivas denominadas de *broadcast(m)* e *deliver(m)*, sendo *m* uma mensagem. A primitiva *broadcast(m)* é utilizada para enviar a mensagem *m* a todos os processos da rede. Uma vez recebida a mensagem, os processos podem utilizar a primitiva *deliver(m)* para entregar *m* à aplicação. Em especial para o caso da difusão confiável, o desafio está justamente em garantir que todos os processos corretos irão entregar a mensagem recebida. Além disso, toda mensagem *m* possui um identificador do processo emissor (denominado de *sender(m)*) e um número de sequência. Estes dois números garantem que toda mensagem seja única.

A difusão confiável deve garantir as seguintes propriedades [Eugster et al. 2004, Correia et al. 2006]:

- Validade: Se um processo correto executa *broadcast* para uma mensagem *m*, então ele irá em algum momento entregar *m*.

- Acordo: Se um processo correto entrega a mensagem m, então todo processo correto em algum momento entrega m.

- Integridade: Todo processo correto entrega uma mensagem m uma única vez e somente se m foi enviada por um processo correto.

As duas primeiras propriedades garantem que uma mensagem enviada por um processo correto é, em algum momento, entregue por todos os processos corretos da rede. A última propriedade, por sua vez, garante que não serão entregues mensagens provenientes de fora da rede [Hadzilacos and Toueg 1994]. Sendo assim, se o emissor de uma mensagem m falhar, a especificação da difusão confiável garante que a mensagem m será entregue por todos processos corretos ou por nenhum deles.

2.4.1 Difusão Confiável e Atômica

Enquanto a difusão confiável garante a entrega a todos os processos corretos, nenhuma afirmação pode ser feita com relação à ordem em que as mensagens são entregues. Por exemplo, considere as mensagens $m_1^{p_1}$ e $m_2^{p_1}$ transmitidas por difusão pelo processo p_1 para os processos p_2 e p_3. É possível que o processo p_2 receba primeiro $m_1^{p_1}$ e depois $m_2^{p_1}$ enquanto que o processo p_3 receba primeiro $m_2^{p_1}$ e depois $m_1^{p_1}$.

Quando a difusão confiável é feita de maneira ordenada entre todos os processos, *i.e.*, todos os processos entregam todas as mensagens exatamente na mesma ordem, dizemos que a difusão é atômica (conhecida também como *atomic broadcast*, ou *atomic reliable broadcast*, ou ainda *total-order broadcast*). Diversas aplicações utilizam a difusão atômica [Défago et al. 2004].

Como descrito acima, a difusão atômica requer que todo processo correto entregue as mensagens na mesma ordem. De maneira mais formal, a difusão atômica é uma difusão confiável e que também satisfaz à seguinte propriedade [Hadzilacos and Toueg 1994]:

- Ordem Total: Se dois processos corretos p e q entregam duas mensagens m e m', então p entrega m antes de m' se e somente se q entrega m antes de m'.

Sendo assim, a propriedade de ordem total e de acordo garantem que todo processo correto entrega a mesma sequência de mensagens. Existem várias maneiras de implementar a difusão atômica de forma a garantir que todos os processos entreguem a mesma sequência de mensagens. Em geral, todas envolvem um acordo prévio (consenso) da sequência das mensagens a serem entregues.

2.4.2 Outras Estratégias de Ordenação

A difusão atômica apresentada na Seção 2.4.1 garante que todos os processos corretos entregam o mesmo conjunto de mensagens na mesma ordem. Porém, qualquer ordem pode ser adotada. Visto que a difusão atômica independe do emissor da mensagem [Défago et al. 2004], *i.e.*, as mensagens não são entregues na ordem em que foram enviadas, existem restrições adicionais que envolvem também o emissor da mensagem. Neste sentido, existem duas propriedades denominadas de ordem FIFO (*First-In First-Out*) e ordem causal que são descritas a seguir.

Ordem FIFO

Em [Hadzilacos and Toueg 1994] os autores afirmam que, em geral, as mensagens possuem um contexto e que sem ele a mensagem poderia ser mal interpretada. Como exemplo, considere um sistema de reservas de passagens aéreas. O contexto de uma mensagem cancelando uma determinada reserva consiste de uma mensagem anterior cujo contexto era a reserva da passagem. Ou seja, a mensagem de cancelamento não pode ser entregue antes de entregar a mensagem que cria a reserva [Hadzilacos and Toueg 1994].

Sendo assim, a difusão FIFO é uma difusão confiável que garante a propriedade de ordenação FIFO:

- Ordem FIFO: Se um processo correto executa *broadcast* para uma mensagem m antes de executar *broadcast* para a mensagem m', então nenhum processo correto entrega m' antes de entregar m.

É importante notar que a difusão atômica pode garantir que, além de todos os processos entregarem as mensagens em uma mesma ordem, essa ordem seja a ordem FIFO. Este tipo de comunicação satisfaz às propriedades de ordem total e ordem FIFO e é denominado de difusão atômica FIFO.

Ordem Causal

Os algoritmos de difusão apresentados até então consideram a ordem apenas definida pelo remetente ou a ordem total. Por outro lado, uma mensagem m pode depender de outras mensagens enviadas por outros processos. Neste caso, nenhum algoritmo apresentado até agora satisfaz esse requisito. Como exemplo, considere uma rede social onde os usuários transmitem mensagens por difusão confiável FIFO. Neste cenário, a seguinte situação pode acontecer: usuário A realiza uma publicação e o usuário B envia uma resposta para esta publicação. Esta resposta só pode ser compreendida por usuários que tenham visualizado a publicação de A. Um usuário C não pode receber a resposta de B antes de receber a publicação de A.

Neste tipo de cenário surge a necessidade de criar um outro tipo de ordenação que leva em consideração eventos com precedência causal. Esse conceito de precedência causal foi formalizado por Lamport [Lamport 1978] e é denotado pela relação de precedência representada pelo símbolo $\longrightarrow$. A relação de precedência é definida a seguir [Défago et al. 2004]:

- **Definição:** *considere e_i e e_j dois eventos de um sistema distribuído. A relação $e_i \longrightarrow e_j$ procede se uma das três condições descritas a seguir for satisfeita:*

  1. *e_i e e_j são eventos do mesmo processo, onde e_i ocorreu antes de e_j, tendo como referência o relógio local;*
  2. *e_i é a transmissão de uma mensagem m e e_j é o recebimento de m por outro processo;*
  3. *Há um terceiro evento e_k, tal que, $e_i \longrightarrow e_k$ e $e_k \longrightarrow e_j$.*

Se pelo menos uma destas três condições for satisfeita, então é possível definir a difusão causal como uma difusão confiável que satisfaz à seguinte propriedade:

- Ordem Causal: se existe uma relação de precedência entre a mensagem m e a mensagem m', então nenhum processo correto entrega m' antes de entregar m.

Assim como na difusão atômica FIFO, é possível incorporar a difusão causal na difusão atômica. Sendo assim, todas as mensagens transmitidas terão a mesma ordem, respeitando ainda a precedência causal. Ou seja, a difusão atômica causal respeita as propriedades de ordem total e ordem causal.

A ordem causal pode ser obtida através da combinação da ordem FIFO com a propriedade a seguir [Hadzilacos and Toueg 1994]:

- Ordem Local: se um processo envia uma mensagem m e algum processo entrega m antes de transmitir m', então nenhum outro processo correto entrega m' antes de entregar m.

3 Funções Virtualizadas de Rede

Este capítulo introduz os principais conceitos de Virtualização de Funções de Rede (*Network Function Virtualization* - NFV) utilizados no decorrer deste trabalho. A Seção 3.1 apresenta a motivação e as vantagens para utilização de NFV, enquanto que a Seção 3.2 descreve a padronização de uma arquitetura para o gerenciamento e orquestração das Funções Virtualizadas de Rede (*Virtualized Network Function* - VNFs).

3.1 Introdução às Funções Virtualizadas de Rede

As redes de computadores são constituídas de diversos serviços, muitos dos quais são implementados nos chamados *middleboxes* (*e.g.*, *firewall*, *Network Address Translation*, sistemas de detecção de intrusão) e disponibilizados, portanto, como dispositivos de hardware [Sekar et al. 2012]. Estes serviços representam uma importante fração das despesas de capital (*CAPital EXpenditures* - CAPEX) e despesas operacionais (*OPerational EXpenditures* - OPEX) de uma rede, já que cada um desses componentes requer hardware especializado [Cotroneo et al. 2014]. Além disso, muitos destes serviços de rede produzidos em hardware são complexos para gerenciar e solucionar problemas [Sherry et al. 2012]. Por fim, pode existir um desperdício dos recursos disponíveis no sistema, uma vez que os *middleboxes* não fornecem meios para ajustar seus recursos, de maneira dinâmica, de acordo a demanda. Por exemplo, se um determinado *middlebox* está utilizando metade dos recursos disponíveis, estes poderiam ser alocados para outra tarefa.

O conceito de NFV surge para solucionar estes problemas, simplificando o projeto, desenvolvimento e gerência dos diversos serviços de rede [Chiosi et al. 2012]. Através de técnicas de virtualização, serviços de rede são disponibilizados por meio de dispositivos virtuais, denominados de VNFs, que são executados em hardware genérico (*e.g.*, arquitetura x86). Dessa forma, a inclusão de novos serviços, bem como a alteração e atualização de serviços já existentes, não necessitam da instalação de equipamentos de hardware especializados, diminuindo custos e flexibilizando o provisionamento de serviços [Mijumbi et al. 2016].

A Figura 3.1 mostra a comparação entre uma rede tradicional e um ambiente NFV. Serviços de rede em uma rede tradicional podem ser implementados e executados utilizando uma camada de virtualização. Essas funções são iniciadas sob demanda e removidas quando não são mais necessárias. Além disso, são mais fáceis de serem gerenciadas e operadas [Turchetti and Duarte 2015].

A abordagem baseada em NFV apresenta diversas vantagens em comparação com as redes tradicionais, tais como a flexibilidade para provisionamento de serviços e a redução de OPEX e de CAPEX [Cotroneo et al. 2014]. Além da redução de despesas para a aquisição e manutenção de equipamentos, há vantagens como a economia de energia e de espaço físico [Han et al. 2015]. Além disso, a flexibilidade do gerenciamento da rede também melhora, na medida em que serviços de rede são facilmente instanciados, utilizados e removidos. Por fim, as

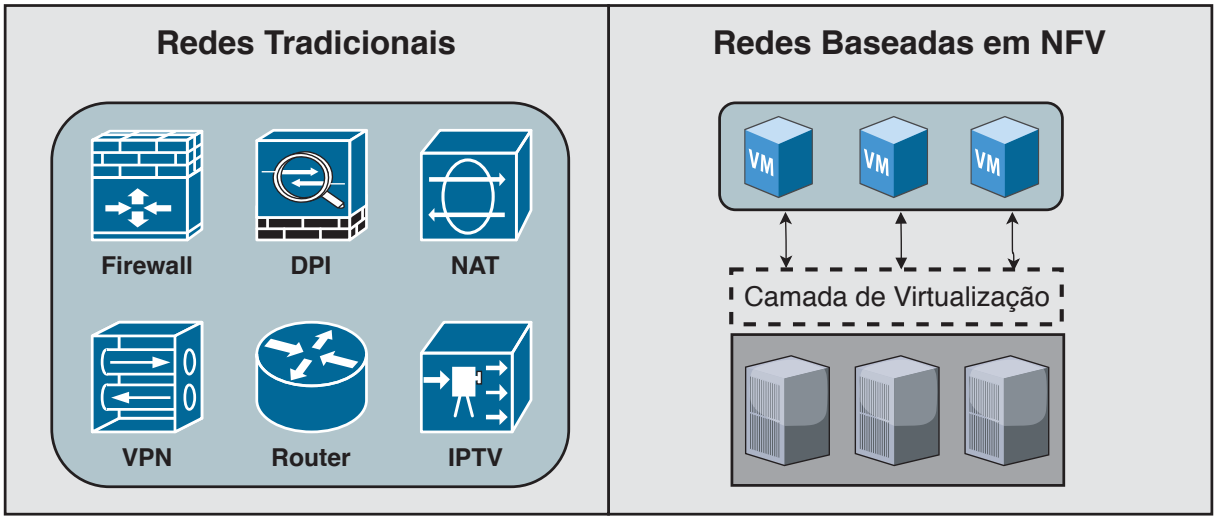


Figura 3.1: Dispositivos de hardware sendo disponibilizados em um ambiente NFV.

VNFs podem ter seus recursos ajustados de maneira automática, aumentando ou diminuindo de acordo com a demanda necessária, fazendo uso eficiente dos recursos do sistema.

Apesar das grandes melhorias que a tecnologia NFV promete, existem diversos desafios a serem estudados para proporcionar um ambiente confiável para a execução destes serviços de rede [Han et al. 2015]. Um destes desafios é manter o desempenho das VNFs próximo ao desempenho dos dispositivos produzidos em *middleboxes*. Existem diversos trabalhos que oferecem meios para desenvolver VNFs de maneira eficiente, utilizando, por exemplo, aceleradores de pacotes [Kourtis et al. 2015], sistemas operacionais minimalistas [Martins et al. 2014] e plataformas de alto desempenho[Hwang et al. 2015]. Outro grande desafio é como realizar, de maneira eficiente, a migração da infraestrutura da rede já existente para a solução baseada em NFV, uma vez que existe um grande acoplamento entre seus componentes.

3.2 O Padrão ETSI para o Gerenciamento e Orquestração de Funções Virtualizadas de Rede

Com o objetivo de padronizar uma arquitetura para plataformas NFV, o grupo ISG (*Industry Specification Group*) criado pela ETSI (*European Telecommunications Standards Institute*) vem coordenando esforços para propor uma arquitetura padrão para NFV [ETSI 2016]. Esta arquitetura tem como objetivo padronizar a execução e gerência dos serviços oferecidos em um ambiente NFV, permitindo o suporte à execução de VNFs provenientes de diferentes desenvolvedores [ETSI 2015]. Essa arquitetura também facilita a integração entre diferentes plataformas NFV e sistemas legados.

A Figura 3.2 ilustra a arquitetura NFV em alto nível proposta pela ETSI, a qual é composta por três blocos principais: NFV-MANO (NFV *MANagement and Orchestration*), NFVI (NFV *Infrastructure*) e as VNFs. Cada bloco é descrito a seguir.

O bloco NFV-MANO possui os módulos operacionais de controle e orquestração de VNFs, responsáveis pelo ciclo de vida e gerenciamento de recursos das funções virtualizadas. As responsabilidades deste bloco vão desde a criação de VNFs até a composição entre elas. Além disso, o NFV-MANO fornece interfaces de comunicação padronizadas e abstrai os recursos computacionais necessários para a execução de VNFs. Os módulos que compõe o NFV-MANO serão descritos adiante.

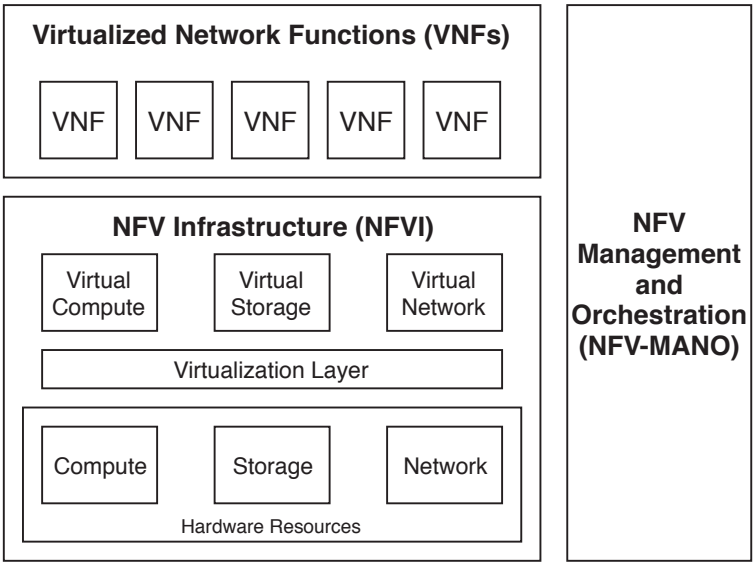


Figura 3.2: Arquitetura NFV alto nível proposta pela ETSI.

O segundo bloco ilustrado na Figura 3.2 é a NFVI. A NFVI representa a infraestrutura virtualizada onde as VNFs serão instanciadas, gerenciadas e executadas. A parte física da NFVI consiste de recursos computacionais (*Compute*), armazenamento (*Storage*) e de rede (*Network*). A NFVI realiza a abstração de recursos físicos em recursos virtuais através de uma camada de virtualização, que é composta por um *hypervisor* que cria dispositivos virtualizados como máquinas virtuais [Sahoo et al. 2010] ou como *containers* [Docker 2017]. Isso permite que cada VNF seja executada de maneira independente e isolada. Além disso, a arquitetura permite que existam múltiplas NFVIs abrangendo vários locais (*e.g.*, servidores separados geograficamente, múltiplos *data centers*). Isso permite que o NFV-MANO possa escolher, através de algoritmos de *placement*, o melhor local de instanciação de determinada VNF, de forma a obter o melhor desempenho daquela função.

Por fim, o terceiro bloco da arquitetura são as *Virtualized Network Functions* (VNFs). Este bloco representa as instâncias de cada elemento operacional virtualizado responsável pela execução das funções de rede que estão executando sobre a NFVI. Em outras palavras, as VNFs são funções de rede capazes de executar em uma infraestrutura virtualizada e que possuem interfaces de comunicação bem definidas para serem gerenciadas pelo NFV-MANO. Cada VNF possui um VNF *Descriptor* (VNFD) associado. Um VNFD é um *template* responsável por especificar uma VNF em termos de requisitos operacionais e de implantação [ETSI 2014]. Estes descritores também definem outros aspectos de uma VNF, como o versionamento de software, configurações de rede (*e.g.*, Endereço IP, interfaces de rede), e recursos computacionais (*e.g.*, CPU, RAM, disco). Além disso, os VNFDs possuem informações que são utilizadas pelo NFV-MANO, como políticas para a realocação de recursos computacionais sob demanda e de monitoramento. Em geral, os VNFDs são baseados no padrão TOSCA (*Topology and Orchestration Specification for Cloud Applications*) [Li and Crandall 2017], que é normalmente descrito em formato YAML (*YAML Ain't Markup Language*) ou JSON (*JavaScript Object Notation*). Além do VNFD, cada VNF deve fornecer um EM (*Element Management*), módulo responsável pelo FCAPS (*Fault, Configuration, Accounting, Performance, Security, Management*) de cada função virtualizada.

Conforme descrito acima, o bloco NFV-MANO realiza a gerência das funções virtualizadas de rede. A Figura 3.3 mostra os detalhes do bloco NFV-MANO, que pode ser dividido em três módulos principais: NFV *Orchestrator* (NFVO), VNF *Manager* (VNFM) e *Virtualized Infrastructure Manager* (VIM).

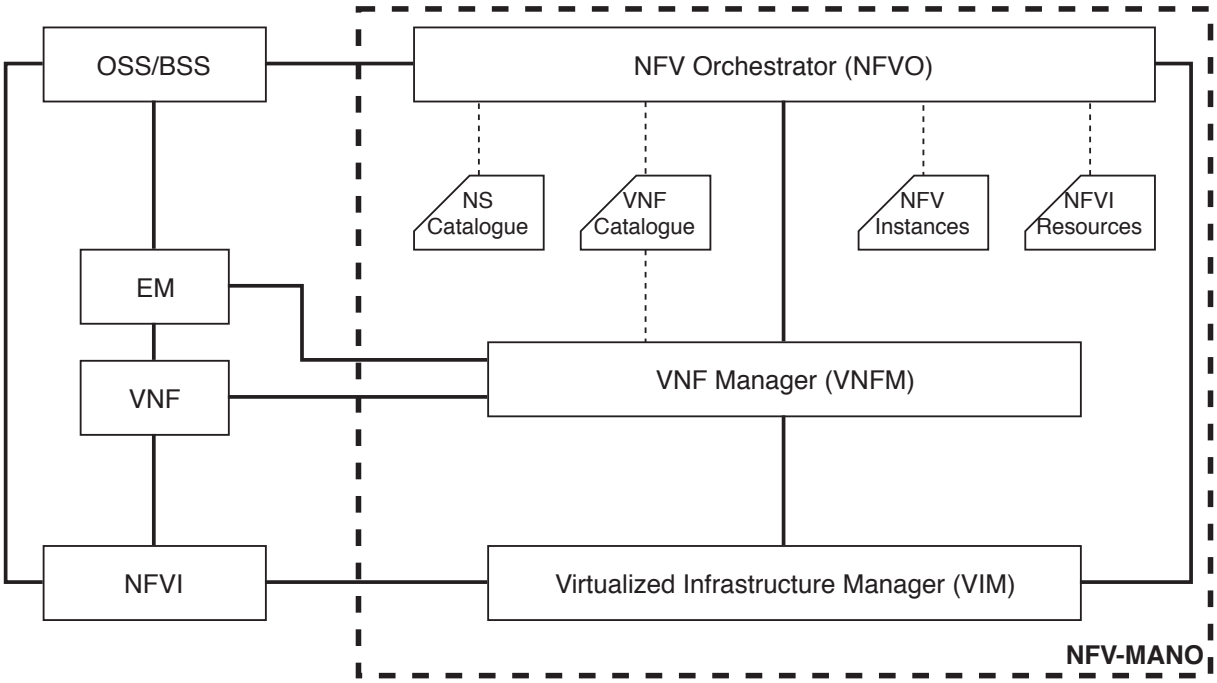


Figura 3.3: Bloco NFV-MANO detalhado.

De acordo com a arquitetura proposta pela ETSI, o NFVO possui duas principais tarefas: gerenciamento e orquestração dos recursos alocados na infraestrutura virtualizada; gerência do ciclo de vida dos serviços de rede. Dessa forma, o NFVO coordena a composição entre VNFs, formando um *Service Function Chain* (SFC). Para estas tarefas, o NFVO utiliza os módulos VNFM e VIM, descritos adiante. Além disso, o NFVO realiza outras tarefas, como o aumento ou diminuição de recursos de uma VNF de acordo com a demanda, gerenciamento de políticas, validação e autorização de requisições na NFVI, entre outras.

O módulo VNFM por sua vez tem como responsabilidade principal a realização da gerência do ciclo de vida das VNFs. A arquitetura prevê a utilização de um ou mais VNFMs, porém as operações de gerência devem ser suportadas por qualquer VNF, independe da sua implementação [ETSI 2014]. Entre as atribuições deste módulo estão: instanciação, remoção, atualização, configuração, atualização e escalonamento das VNFs; monitoramento dos recursos utilizados; notificações de falhas. O VNFM também deve ter acesso aos repositórios e às diferentes versões de VNFs, além de total conhecimento dos descritores das mesmas. Para executar suas funcionalidades, o VNFM faz uso dos descritores, como o VNF *Descriptor*.

O terceiro e último módulo do NFV-MANO é o VIM, responsável por realizar o controle e gerenciamento dos recursos computacionais da infraestrutura virtualizada. Algumas das tarefas executadas pelo VIM compreendem: criação de dispositivos virtuais (*e.g.*, máquinas virtuais, *containers*); modificação dos recursos computacionais alocados; monitoramento da infraestrutura; gerenciamento das imagens virtuais (*i.e.*, *template* que contém o sistema operacional, arquivos e aplicações). Assim como o VNFM, a arquitetura proposta pela ETSI também prevê o uso de múltiplos VIMs, onde cada VIM gerencia uma NFVI. O OpenStack [OpenStack 2017] é uma plataforma clássica que implementa o VIM.

Finalmente, apesar de não pertencer a arquitetura NFV descrita, o módulo OSS/BSS (*Operations Support System*/*Business Support System*) representa sistemas e aplicações externas que comunicam-se com o NFV-MANO. Uma vez que o conceito de NFV recomenda o desenvolvimento de plataformas abertas, de forma a permitir a integração de soluções de diferentes desenvolvedores, os sistemas existentes não podem gerenciar as VNFs diretamente,

visto que estas VNFs podem conter soluções proprietárias. Assim, o módulo OSS/BSS comunica-se com o NFV-MANO para realizar as requisições de gerência (*e.g.*, instanciar, remover, atualizar VNFs) através de interfaces padronizadas.

4 Simplificando o Gerenciamento do Ciclo de Vida de Funções Virtualizadas de Rede

Um dos principais desafios da Virtualização de Funções de Rede é o gerenciamento do ciclo de vida das VNFs. Atualmente, as soluções para o gerenciamento de VNFs são complexas e necessitam de um entendimento profundo da infraestrutura subjacente. Neste capítulo é proposta uma arquitetura para a gerência do ciclo de vida das VNFs que fornece compatibilidade entre diferentes plataformas e cenários de utilização de VNFs, simplificando as operações de gerência na medida em que diminui a necessidade de que o operador das funções conheça detalhes da infraestrutura virtualizada. Um protótipo, denominado de *vCommander* foi implementado e resultados obtidos em um ambiente experimental demonstram a efetividade da arquitetura proposta na realização de todas as operações do gerenciamento do ciclo de vida de VNFs.

Este capítulo está organizado da seguinte forma. A Seção 4.2 apresenta uma breve fundamentação sobre plataformas de gerência e orquestração de NFV. A Seção 4.3 apresenta a arquitetura conceitual da nossa proposta. A Seção 4.4 descreve a implementação do protótipo *vCommander* e resultados experimentais são apresentados na Seção 4.5. Por fim, a Seção 4.6 apresenta as conclusões e os trabalhos futuros.

4.1 Justificativa da Proposta

Atualmente, diversas plataformas implementam o VNFM proposto pela ETSI [Tacker 2017, OpenBaton 2017, ETSI 2017, ONAP 2017, Rift.io 2017]. Em geral, estas plataformas demandam uma série de procedimentos para as operações do ciclo de vida das VNFs, que vão desde a criação de descritores de máquinas virtuais (*Virtual Machines* - VMs) até a configuração manual do software que irá executar a função virtualizada. Dessa forma, tais soluções se tornam complexas e exigem do operador das funções um grande entendimento da infraestrutura do sistema [Shen et al. 2015]. Além disso, as plataformas disponíveis são pouco flexíveis, fazendo com que a integração com outras tecnologias exija um grande esforço dos desenvolvedores para adaptar as ferramentas utilizadas. Por fim, algumas destas soluções não gerenciam VNFs de maneira completa, sendo necessária a utilização de ferramentas adicionais ou gerenciamento manual das funções em nível de software.

Para suprir tais deficiências, é proposta a arquitetura de um VNF *Manager* que simplifica o gerenciamento do ciclo de vida das VNFs. O VNFM proposto define uma API que permite o controle do ciclo de vida das funções virtualizadas através de requisições feitas pelo usuário. Além disso, são definidas outras duas APIs que flexibilizam e simplificam a compatibilidade do VNFM com outras plataformas e cenários no ambiente NFV. Isso reduz o conhecimento necessário da infraestrutura virtualizada por parte do operador das funções e simplifica os

procedimentos de gerência. É proposto também um submódulo do VNFM que utiliza todas as APIs definidas para controlar de maneira autônoma todo o ciclo de vida das VNFs, desde o nível de hardware até o nível de software.

Como prova de conceito da arquitetura proposta, é apresentado um protótipo, denominado de *vCommander*, que implementa a arquitetura do VNFM proposto. Em especial, é descrito como o *vCommander* utiliza o *Event Manager* e as APIs de comunicação para implementar a inserção, remoção e atualização das VNFs. Resultados obtidos em um ambiente experimental comprovam a efetividade do *vCommander* na realização de todas as operações de gerenciamento. Em especial, foi analisado o desempenho individual de cada operação.

4.2 Plataformas de Gerenciamento de Funções Virtualizadas de Rede

Diferentes soluções capazes de realizar funções destacadas no módulo de gerência e orquestração de VNFs, de acordo com as especificações ETSI, foram propostas [ETSI 2017, Tacker 2017, OpenBaton 2017, ONAP 2017, Rift.io 2017]. Apesar da diversidade de plataformas que implementam parcialmente responsabilidades do NFV-MANO, nenhuma é totalmente adequada ao conjunto de operações previstas para este bloco.

O Tacker [Tacker 2017], por exemplo, é uma plataforma capaz de executar funções de VNFM e NFVO, abstraindo a utilização do VIM (*e.g.*, OpenStack [OpenStack 2017]) com funções de mais alto nível. Entre as funções implementadas relacionadas ao VNFM estão o controle do ciclo de vida básico, monitoramento, escalonamento e simplificação da configuração inicial das VNFs. O módulo que atua como NFVO é capaz de aplicar políticas de instanciação e escalonamento, bem como manipular estruturas complexas de serviços de rede, descritas através de SFCs [Halpern and Pignataro 2015]. Apesar de implementar grande parte das funções previstas para o NFV-MANO, o Tacker não conta com uma interface gráfica padronizada, módulos de simulação e nem mesmo com a possibilidade de implantação de algoritmos de *placement* otimizados [Mechtri et al. 2017].

O Open Source MANO (OSM) [ETSI 2017] consiste da implementação dos três módulos operacionais do NFV-MANO (*i.e.,* NFVO, VNFM e VIM) para realizar a configuração e abstração de VNFs, orquestração de recursos e elementos para controle da infraestrutura. Entre as operações suportadas pelo OSM estão um serviço de orquestração responsável por controlar diversos aspectos do ciclo de vida das VNFs e possibilitando a execução de serviços complexos, suporte a diferentes VIMs, controladores SDN e ferramentas de monitoramento. Além disso, conta com um VNFM genérico que possibilita a integração a VNFMs específicos, porém, exigindo modificações no seu orquestrador de serviço para total compatibilidade [Dmitriy Andrushko 2017].

A plataforma Open Baton [OpenBaton 2017] também implementa um NFV-MANO baseado na especificação do ETSI. A plataforma consiste de um NFVO e um VNFM e possui suporte nativo a diferentes VIMs. Além do monitoramento e controle de ciclo de vida das VMs que executam as funções de rede, o Open Baton utiliza rotinas definidas através de um VNF *package* com intuito de configurar a função de rede propriamente dita, ou seja, é capaz de alterar internamente o estado do *software* executado pela VM. Apesar de ser capaz de suportar VIMs heterogêneos e apresentar uma interface de usuário amigável, o Open Baton não apresenta nenhum suporte nativo para instanciação e gestão de SFCs [Mechtri et al. 2017].

De maneira geral, as plataformas existentes implementam funções destinadas ao VNFM e NFVO, além de suportarem a utilização de VIMs externos (*e.g.*, OpenStack, OpenVIM [OpenVIM 2017]). Entretanto, algumas funcionalidades destacadas para esses blocos operacio-

nais de orquestração e gerência das VNFs, como a possibilidade de realizar alterações internas a VNF (*i.e.*, na função de rede em si) ou a aplicação e verificação de políticas complexas, não contam com suporte nativo.

4.3 Uma Arquitetura para a Gerência do Ciclo de Vida das VNFs

Nesta seção é apresentada a arquitetura conceitual e são detalhados os principais componentes do VNFM proposto. Em seguida, são descritas as interfaces necessárias para a comunicação entre os módulos da solução. Por fim, são detalhadas as funcionalidades de cada uma das APIs propostas.

4.3.1 Componentes da Arquitetura

Este trabalho introduz uma arquitetura do módulo VNFM presente no componente de gerenciamento e orquestração de funções virtualizadas de rede. Foram definidas três APIs que são utilizadas para gerenciar o ciclo de vida das VNFs, fornecendo maior flexibilidade para o desenvolvimento de um VNFM. Em especial, é proposto um submódulo do VNFM, denominado de *Event Manager*, responsável por interagir com estas APIs. Os requisitos de cada uma das APIs definidas são descritos na Subseção 4.3.2.

O módulo VNFM é um subsistema de gerência responsável por controlar o ciclo de vida das VNFs. O VNFM permite a instanciação, remoção e atualização das máquinas virtuais que hospedam as VNFs, bem como a gerência e monitoramento das VNFs em nível de função, *i.e.*, a configuração e execução do software que executa a funcionalidade da VNF. O VNFM proposto é capaz de gerenciar as VNFs instanciadas independentemente dos seus propósitos (*e.g.*, segurança, desempenho e encaminhamento).

Em geral, as plataformas que oferecem mecanismos para gerenciar o ciclo de vida de VNFs demandam uma série de procedimentos exaustivos, como por exemplo, a criação e remoção de descritores de máquinas virtuais, configuração e instanciação em nível de software da VNF. Além disso, é exigida do usuário uma grande quantidade de informações, referentes ao funcionamento do sistema [Shen et al. 2015]. Estas informações incluem detalhes de configuração da rede até parâmetros específicos do sistema. Neste sentido, o VNFM proposto define uma API que permite realizar a gerência das VNFs através de requisições geradas pelo usuário. Como exemplo, considere a instanciação de uma VNF. Para realizar essa tarefa em outras plataformas é comum a necessidade de executar uma série de comandos que envolvem o registro de descritores (*e.g., VNF Descriptors*) e a instanciação da máquina virtual onde a função irá ser executada. Ainda, algumas plataformas não instanciam a função da VNF, sendo necessária a execução manual ou o desenvolvimento de ferramentas adicionais que realizem a execução da função de rede. Ao utilizar o VNFM proposto, uma única requisição é suficiente para que a VNF solicitada seja instanciada e executada.

A arquitetura proposta é apresentada na Figura 4.1. Esta arquitetura consiste de duas camadas: a camada de usuário (*User Layer*) e a camada NFV (*NFV Environment*). Na camada de usuário, o operador se comunica com módulos providos por plataformas genéricas que ofereçam serviços em nível de usuário (*e.g.,* Marketplaces [Xilouris et al. 2014] e ferramentas para visualização em NFV [Franco et al. 2016]). Tais plataformas se comunicam com os módulos disponíveis no NFV-MANO através de uma API de comunicação e podem se beneficiar dos recursos disponíveis, tais como velocidade e facilidade na instanciação de VNFs, monitoramento da operação das VNFs e gerenciamento eficaz de recursos. O *NFV Environment* consiste de três componentes principais: NFV-MANO, NFVI e as VNFs. Conforme descrito no Capítulo 3, o

componente NFV-MANO é responsável por gerenciar e orquestrar as funções virtualizadas de rede. O NFVI por sua vez representa a infraestrutura virtualizada onde as VNFs irão executar. Nesta infraestrutura estão presentes os recursos de processamento, armazenamento e de rede. Por fim, as VNFs representam as instâncias das funções virtualizadas de rede.

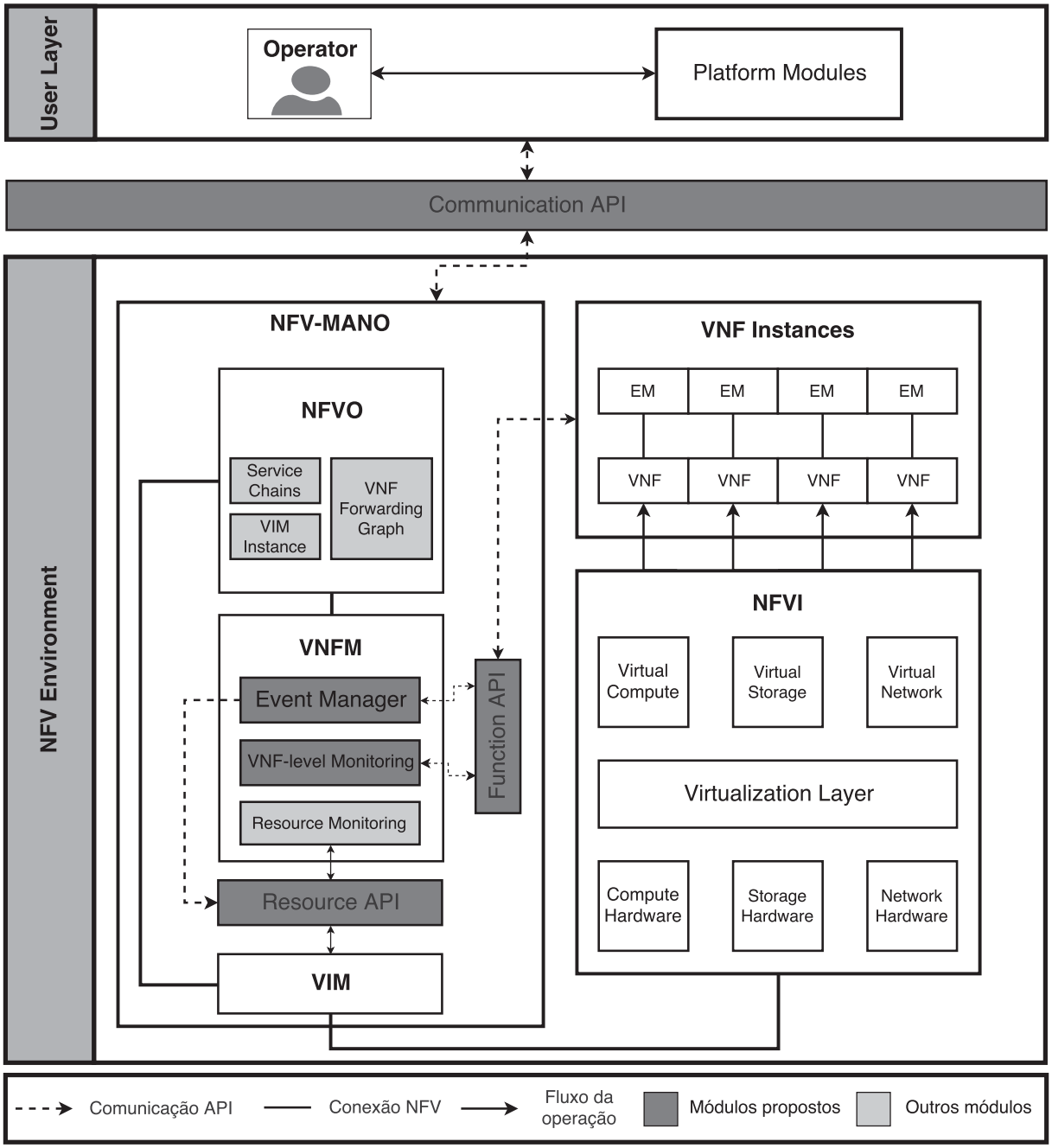


Figura 4.1: Arquitetura onde o VNFM está inserido.

Para prover a automação necessária foi implementado um módulo principal denominado *Event Manager* e um módulo de monitoramento denominado *VNF-level Monitoring*. O *Event Manager* é o módulo principal do VNFM responsável por receber diretamente as requisições originadas da camada de usuário. A partir destas requisições, o VNFM realiza, de maneira automática, todos os procedimentos necessários para a instanciação da VNF, desde a criação da VM até a execução da função. Os detalhes das funcionalidades da gerência fornecidas pelo *Event Manager* são descritos na Subseção 4.4.1. Para o monitoramento das funções das VNFs,

um módulo especializado é introduzido. O *VNF-level Monitoring* é responsável por monitorar a função propriamente dita de cada VNF. Neste caso, as métricas coletadas são relacionadas ao uso da função, como por exemplo, a quantidade de pacotes processados e a latência das operações. Por fim, o módulo *Resource Monitoring* monitora as máquinas virtuais de cada VNF instanciada. Neste caso, as métricas de monitoramento são relacionadas aos recursos utilizados de cada VNF, tais como uso de CPU, memória e disco. Uma vez obtidas estas métricas é possível realizar rotinas para o gerenciamento das VNFs (*e.g.,* redistribuição de recursos sob demanda e recuperação de falhas).

4.3.2 Interfaces de Comunicação

A arquitetura proposta descrita na Subseção 4.3.1 utiliza três APIs para simplificar e flexibilizar as operações das funções virtualizadas. Uma destas APIs é a *Management API*, a qual o VNFM utiliza para receber requisições da camada de usuário. As outras duas APIs são denominadas de *Resource* e *Function API* e são utilizadas para gerenciar o ciclo de vida das VNFs tanto em nível de hardware quanto em nível de software. Cada uma destas APIs são descritas a seguir.

Com o objetivo de facilitar a interação do usuário com o controle do ciclo de vida das VNFs, a *Management API* define um conjunto de funções que permite ao usuário realizar operações sobre as funções virtualizadas. Estas funções não exigem do usuário a execução de vários procedimentos e evitam a necessidade de especificar múltiplos parâmetros do sistema. Todas as funcionalidades disponibilizadas pela *Management API* estão descritas na Tabela 4.1.

Tabela 4.1: Funcionalidades da *Management API*.

Função	**Descrição**
vnf_create(vnfd, vnf_function)	Cria uma VNF
vnf_delete(vnf_id)	Remove uma VNF
vnf_update(vnf_id, vnfd)	Atualiza os recursos de uma VNF
vnf_function_update(vnf_id, vnf_function)	Atualiza a função de uma VNF
sfc_create(vnffgd)	Cria um *Service Function Chaining*

Uma vez recebidas requisições pela *Management API* da camada de usuário, o VNFM comunica-se com vários outros módulos que integram a arquitetura NFV definida pela ETSI. Um destes módulos é o VIM , subsistema que gerencia os recursos e controla as interações da VNF com o ambiente de virtualização. Já o EMS é responsável pelo FCAPS das VNFs [ETSI 2014].

As comunicações entre o VNFM e os demais módulos durante o controle do ciclo de vida das VNFs são realizadas por meio de requisições REST (*REpresentational State Transfer*) ou através de um *Message Broker* (*e.g.,* RabbitMQ). Em geral, estas implementações são pouco flexíveis, onde cada plataforma de gerência de VNFs fornece uma solução que implementa o conjunto de funcionalidades necessárias para comunicação do VNFM com estes módulos. Dessa maneira, utilizar outras tecnologias torna-se uma tarefa complexa, exigindo um grande esforço dos desenvolvedores para adaptar as ferramentas utilizadas. Nesse sentido, propomos duas APIs para simplificar a compatibilidade e flexibilidade do VNFM. Estas APIs são denominadas de *Resource API* e *Function API* e são descritas a seguir.

A *Resource API* define as funcionalidades necessárias para que o VNFM possa gerenciar os recursos das máquinas virtuais que irão hospedar as VNFs. A Tabela 4.2 descreve o conjunto mínimo de funções que deve ser implementado nesta API. Novas funções podem ser adicionadas

à *Resource API* (*e.g.,* remoção e atualização de VNFD), conforme a necessidade do operador. Em geral, esta API é fornecida através de um *plugin* ou *driver* do próprio VIM.

Tabela 4.2: Conjunto de funcionalidades da *Resource API*.

Função	**Descrição**
vnfd_create(vnfd)	Cria um *VNF Descriptor*
vm_create(vnfd_id)	Cria uma máquina virtual baseada no VNFD
vm_delete(vnf_id)	Remove uma máquina virtual
vm_update(vnf_id, vnfd)	Atualiza uma máquina virtual

A *Function API* por sua vez define as funcionalidades que o VNFM deve executar em nível de função. Esta API é responsável por administrar o FCAPS das VNFs, fazendo o papel de EM. A Tabela 4.3 descreve as funções que devem ser implementadas nesta API. Da mesma forma como na *Resource API*, funções adicionais podem ser incluídas na *Function API*. Em especial, através dessa API pode-se definir métricas customizadas de monitoramento obtidas pelo módulo *VNF-level Monitoring*, como por exemplo, número de pacotes bloqueados em um *firewall*. Em geral, as próprias VNFs oferecem esta interface para que o VNFM possa coletar dados e executar instruções.

Tabela 4.3: Conjunto de funcionalidades da *Function API*.

Função	**Descrição**
write_function(vnf_ip, function)	Insere a função da VNF na máquina virtual
start_function(vnf_ip)	Inicia a função da VNF
stop_function(vnf_ip)	Para a função
get_log(vnf_ip)	Obtém o *log* da função
get_metrics(vnf_ip)	Obtém métricas do uso da função

Uma vez definidas as APIs de comunicação necessárias para gerenciamento do ciclo de vida das VNFs, é possível que diferentes tecnologias sejam utilizadas para implementar tais APIs. Com isso, reduz-se a complexidade para adoção de novas funções e também se facilita a implementação de novos serviços e rotinas em diferentes infraestruturas. Baseando-se na arquitetura proposta, na seção a seguir é apresentada e detalhada a implementação de uma prova de conceito, denominada *vCommander*.

4.4 *vCommander*: Protótipo do VNF *Manager*

Nesta seção é apresentado o *vCommander*, um VNFM implementado como prova de conceito da arquitetura proposta na Seção 4.3. Na Subseção 4.4.1 são detalhadas as operações de criação, remoção e atualização de uma VNF. Em seguida, detalhes do protótipo *vCommander* implementado são descritos na Subseção 4.4.2.

4.4.1 Gerenciando o Ciclo de Vida das VNFs

A partir da *Resource API* e *Function API* é possível realizar as operações de gerência do ciclo de vida das VNFs, desde a criação da máquina virtual, até a instanciação da função propriamente dita. A seguir é descrito como o *vCommander* utiliza estas APIs para gerenciar as

VNFs, detalhando as operações de instanciação, remoção e atualização, as quais são iniciadas a partir de requisições recebidas pela *Management API*. É importante notar que todos os procedimentos descritos são realizados de maneira automática, sem a intervenção do usuário.

Instanciação de uma VNF

Para a instanciação de uma VNF são necessárias duas etapas: *(i)* criação e configuração da máquina virtual (*Resource API*) e *(ii)* execução da função virtualizada (*Function API*). A primeira etapa consiste de dois passos, sendo que o primeiro passo consiste em utilizar a função *vnfd_create(vnfd)* para adicionar o VNFD recebido da *Management API* para o catálogo de VNFs, gerando um identificador único denominado *vnfd_id*. O segundo passo é a chamada da função *vm_create(vnfd_id)* que irá utilizar o hipervisor para criação da máquina virtual baseando-se no VNFD criado anteriormente. Dado que o tempo de criação varia para cada sistema, o *vCommander* inicia um mecanismo de *polling* que verifica a cada intervalo de tempo se a máquina virtual está ativa. O intervalo de tempo padrão de *polling* do *vCommander* é de um segundo, porém este valor é customizável e pode ser ajustado de acordo com o sistema. Quando o *vCommander* detecta que a criação da máquina virtual está pronta, inicia-se a segunda etapa.

A segunda etapa consiste em inserir a função da VNF na máquina virtual, também recebida como parâmetro da *Management API*, através da chamada *write_function(vnf_ip, function)*. Uma vez inserida, a função da VNF é executada através da função *start_function(vnf_ip)* e neste momento está totalmente funcional. Por fim, o *vCommander* adiciona esta VNF ao catálogo de instâncias de VNFs para que ela possa ser monitorada pelos módulos *Resource Monitoring* e *VNF-level Monitoring*. Para cada operação realizada é verificado se ela foi concluída com sucesso, caso contrário, um mecanismo de *rollback* desfaz as modificações realizadas até o momento.

Remoção de uma VNF

O processo de remoção inicia-se quando o *vCommander* recebe um identificador único da VNF (*vnf_id*) pela *Management API* e posteriormente remove a máquina virtual correspondente através da função *vm_delete(vnf_id)*. Da mesma forma como na instanciação, um mecanismo de *polling* é iniciado para verificar, a cada intervalo de tempo, se a máquina virtual foi removida com sucesso. Em seguida, a VNF é removida do catálogo de VNFs instanciadas e não é mais monitorada pelos módulos de monitoramento. Por fim, o VNFD da VNF é removido do catálogo.

Atualização de uma VNF

A atualização de uma VNF pode ser dividida em dois níveis: *(i)* atualização de recursos (*Resource API*) e *(ii)* atualização da função virtualizada (*Function API*). No caso da atualização de recursos, a máquina virtual que hospeda a VNF pode utilizar a função *vm_update(vnf_id, vnfd)* para atualizar, por exemplo, o número de CPUs, a quantidade de memória RAM e quantidade de disco alocados para a VNF. Essa atualização ocorre da seguinte forma. Pela *Management API*, é recebido o *vnf_id* e o novo descritor da VNF, contendo as modificações que devem ser realizadas. Por exemplo, este descritor pode modificar a quantidade de memória RAM para 512 MB ou alterar o número de CPUs para 8. O *vCommander* executa então a operação de atualização através da *Resource API* e, para concluir as alterações, a máquina virtual é reiniciada.

Para a atualização da função da VNF, o *vCommander* recebe o *vnf_id* e a nova implementação da função virtualizada. Da mesma forma como na instanciação, a nova função é inserida na máquina virtual pela função *write_function(vnf_ip, function)*. Após, a função antiga

é parada através da chamada *stop_function(vnf_ip)* e a nova função é executada pela função *start_function(vnf_ip)*. Note que neste intervalo de tempo é possível que pacotes sejam perdidos.

4.4.2 Protótipo

Como descrito na Seção 4.3, a arquitetura do VNFM proposto consiste dos módulos *Event Manager* e *VNF-level Monitoring* e utiliza as APIs *Management*, *Resource* e *Function* para a comunicação entre os demais componentes. O *Event Manager* foi desenvolvido em Python e é responsável por receber requisições de alto nível da *Management API* e realizar as operações necessárias através da *Resource* e *Function API*. Todas estas requisições são feitas por meio de de requisições REST utilizando a biblioteca Requests[1] e as informações são definidas em JSON. É importante notar que a comunicação entre os módulos pode ser interna (*i.e.,* execução centralizada dos componentes) ou externa (*i.e.,* execução distribuída dos componentes).

O *VNF-level Monitoring* por sua vez, realiza requisições às VNFs a cada intervalo de tempo. O *vCommander* utiliza por padrão um intervalo de tempo de 30 segundos, porém este valor pode ser customizado de acordo com a necessidade. Os dados coletados são enviados para um servidor central, onde é feito o processamento destes dados. Os dados são inseridos no banco de dados InfluxDB[2], que realiza em tempo real o processamento dos dados coletados do uso das funções virtualizadas de rede. Ambos os módulos foram também desenvolvidos em Python.

Para a *Resource API*, todas as funcionalidades descritas na Tabela 4.2 foram implementadas. Esta API deve ser disponibilizada para que o VNFM possa realizar as requisições e gerenciar as máquinas virtuais. Da mesma forma, a *Function API* implementa as funcionalidades descritas na Tabela 4.3. Em especial, a implementação desta API foi feita de maneira genérica, de forma a utilizar a mesma API para diferentes VNFs. Isso é feito passando o endereço IP da VNF como parâmetro da requisição. Neste trabalho, a *Resource API* foi implementada utilizando o Tacker [Tacker 2017]. Para a *Function API*, cada VNF instanciada disponibiliza uma interface REST onde o *vCommander* realiza as requisições. A próxima seção apresenta resultados experimentais do *vCommander* avaliando o tempo e a quantidade de recursos utilizados por cada operação.

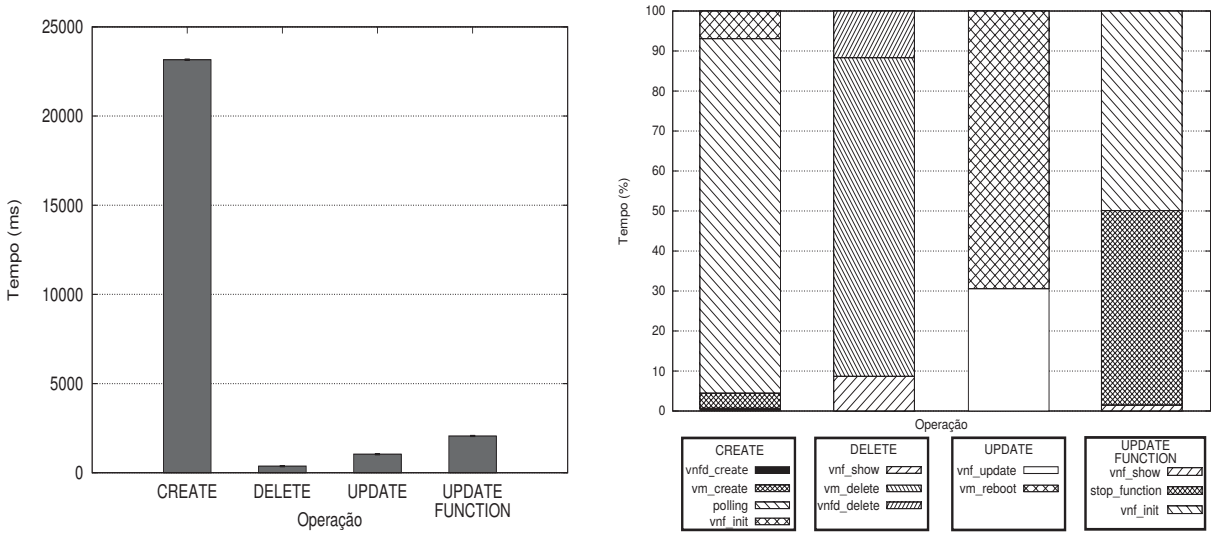
4.5 Avaliação Experimental

Com o objetivo de verificar o comportamento e o desempenho do protótipo *vCommander*, esta seção apresenta resultados experimentais. Na Subseção 4.5.1, o cenário utilizado para os experimentos é apresentado. A seguir, na Subseção 4.5.2, as justificativas para as métricas escolhidas e os resultados obtidos são apresentados e discutidos.

4.5.1 Cenário de Avaliação

Como descrito na Seção 4.4.2 o *vCommander* comunica-se com a *Resource API* para gerenciar os recursos virtuais das VNFs. No protótipo implementado a *Resource API* é disponibilizada pelo Tacker e OpenStack, e sua execução depende de um servidor de virtualização que esteja executando tais serviços. Assim, foi utilizado um servidor Dell PowerEdge com um processador Intel(R) Xeon(R) E3-1220v6 @ 3.00GHz, com 4 núcleos e 8 *threads* e memória de 8GB DDR4 @ 2400Mhz. Este servidor foi configurado com a distribuição Ubuntu 16.04, a

[1]http://docs.python-requests.org
[2]https://www.influxdata.com/



(a) Tempo total de execução das operações.

(b) Tempo de execução das suboperações em porcentagem.

Figura 4.2: Comparando a vazão das soluções.

versão Pike do OpenStack, bem como o protótipo desenvolvido – embora este possa ser executado remotamente.

O protótipo possui 4 funções principais que são disponibilizadas ao usuário: *Create*, *Delete*, *Update* e *Update Function*. Cada operação executa diversas suboperações que são efetivamente responsáveis por se comunicar com o Tacker para controlar a infraestrutura virtualizada.

A avaliação do protótipo foi realizada da seguinte forma: a operação *Create* foi executada instanciando uma VNF do tipo Click-on-OSv [da Cruz Marcuzzo et al. 2017] com um *firewall*, e recursos de 512 MB de RAM, 1 CPU e 1 GB de disco. A seguir, a operação *Update* foi executada, alterando a memória RAM alocada para a VNF de 512 MB para 1 GB, enquanto os demais recursos permaneceram iguais. Após, a operação *Update Function* foi avaliada, alterando a função em execução para um VNF *forwarder*. Por fim, a VNF foi removida utilizando a operação *Delete*. Todos os testes foram realizados 30 vezes e os resultados são apresentados com um intervalo de confiança de 95%.

4.5.2 Resultados e Discussão

Visto que as VNFs devem ser capazes de serem instanciadas, configuradas e escaladas dinamicamente, o tempo de execução destas funções é uma métrica relevante para medição, uma vez que o atraso na execução destas funções pode impactar negativamente o funcionamento da infraestrutura. Desta forma, o protótipo foi instrumentado de forma a obter o tempo de execução total de cada uma das operações, bem como o tempo de execução das diversas suboperações que as compõem.

Como pode ser observado na Figura 4.2(a), a operação *Create* foi a operação com maior duração, sendo que a suboperação de *polling* consome por volta de 90% do tempo total de execução da operação, conforme Figura 4.2(b). Isto ocorre pois, ao enviar a requisição para a criação da VM (suboperação *vm_create*), a suboperação *polling* deve verificar periodicamente e aguardar a infraestrutura terminar o processo de instanciação, para que a função de rede (configuração Click) possa ser configurada na próxima suboperação(*vnf_init*). Já a operação

Update Function precisa realizar o *upload* de uma nova função de rede (uma nova configuração Click) e aguardar a reinicialização da VNF. Por fim, a operação *Update*, após atualizar a descrição da VM, deve aguardar sua reinicialização, enquanto a operação *Delete* envia comandos para que a VM seja removida.

O uso de NFV também depende da utilização de servidores de virtualização de alta densidade, onde pode ocorrer uma escassez de recursos computacionais se houverem muitas VNFs instanciadas. Assim, o consumo de recursos do protótipo deve ser reduzido, considerando que a própria plataforma OpenStack já possui requisitos elevados para sua execução. Desta forma, tanto o consumo de CPU como o consumo de memória do protótipo foram medidos, utilizando a biblioteca *psutil* do *Python*. A Figura 4.3 apresenta a utilização de memória e CPU de cada operação.

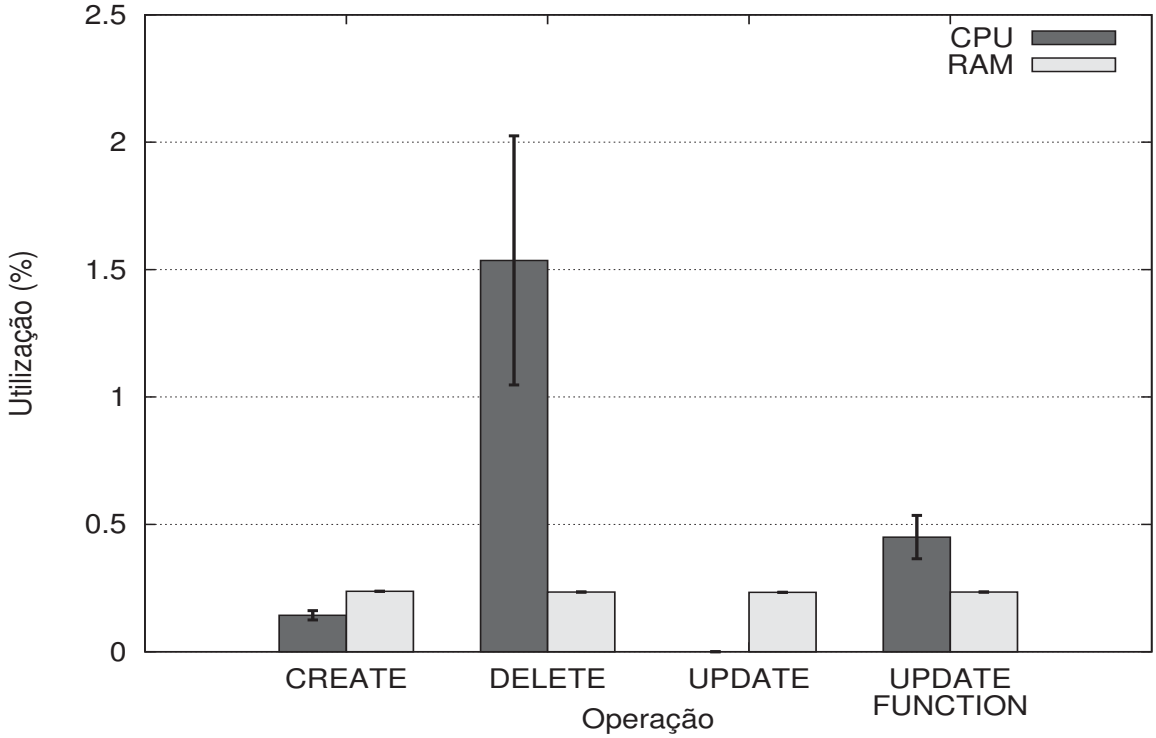


Figura 4.3: Uso de CPU e memória durante as operações.

Com relação a memória RAM, a memória utilizada é ocupada pelas bibliotecas desenvolvidas para a comunicação com a *Resource API* e *Function API* e a gerência do ciclo de vida das VNFs. Como estas bibliotecas são utilizadas por todas as operações, o consumo de memória é semelhante.

Já para a utilização de CPU, é possível perceber que a operação *Delete* apresenta a maior utilização de CPU, enquanto que as operações *Create* e *Update Function*, mesmo possuindo suboperações mais longas, apresentam um menor consumo de CPU. Isto deve-se ao fato de que, no caso da operação *Create*, apesar da suboperação de *polling* ser a mais longa, ela apenas aguarda a resposta da *Resource API*, não realizando nenhum processamento neste intervalo. Já a operação *Update Function* precisa aguardar a VNF reinicializar para finalizar. Desta forma, como a operação *Delete* não é *I/O bound*, não é necessário aguardar nenhum tipo de chamada do sistema, portanto está realizando algum tipo de processamento durante todo o tempo da sua execução. Por fim, a operação *Update* não apresenta um consumo de CPU significativo, já que ela

apenas envia o VNFD atualizado para a *Resource API* e aguarda a reinicialização, sem realizar nenhum tipo de processamento.

Apesar disso, o maior consumo, dentre todas as operações, foi de apenas 1.5% de uso da CPU durante uma duração menor que 1 segundo, enquanto que o consumo de memória RAM corresponde a 0.25% da memória disponível no ambiente testado, de forma que o impacto do protótipo nos recursos do servidor seja mínimo.

4.6 Conclusões Parciais

Neste capítulo foi introduzida uma arquitetura de um VNFM para simplificar o gerenciamento do ciclo de vida das VNFs. Através da utilização de APIs, a arquitetura permite compatibilidade entre diferentes plataformas além de simplificar a gerência do ciclo de vida das VNFs. Além disso, o gerenciamento das funções virtualizadas é feito tanto em nível de hardware quanto em nível de software.

Como prova de conceito foi implementado um protótipo da arquitetura proposta denominado de *vCommander*. Resultados experimentais demonstram que o *vCommander* consegue controlar e facilitar o gerenciamento do ciclo de vida das VNFs consumindo uma quantidade de recursos mínima, utilizando em média menos de 1% de CPU e 0.25% de memória RAM. Além disso, o tempo de operação das principais funcionalidades do *vCommander* foi medido e pode-se observar a efetividade da arquitetura.

Para trabalhos futuros planeja-se estender as funcionalidades da arquitetura proposta para incluir a composição e orquestração de VNFs. Além disso, planeja-se oferecer uma estratégia inteligente de migração de VNFs (*e.g., live migration*) com o objetivo de minimizar a quantidade de pacotes perdidos.

5 Sincronização Consistente de um Plano de Controle Distribuído em Redes SDN

O plano de controle de Redes Definidas por Software (*Software Defined Networks* - SDN) é geralmente centralizado e composto por um único controlador, o que, por definição, traz desafios em termos de disponibilidade, escalabilidade e desempenho. Enquanto um plano de controle distribuído pode resolver estes problemas, alcançar uma sincronização consistente entre múltiplos controladores SDN não é uma tarefa trivial. Neste capítulo é proposta a *VNF-Consensus*, uma VNF que implementa o algoritmo de consenso Paxos para garantir a consistência entre diversos controladores em um plano de controle distribuído. Utilizando essa abordagem, controladores são desacoplados das tarefas de sincronização e podem executar as suas atividades do plano de controle sem a necessidade de executar tarefas computacionalmente custosas, necessárias para manter a consistência.

Para realizar a sincronização das ações entre todos os controladores, cada controlador possui acesso a uma instância da *VNF-Consensus*. Essa instância possibilita que um controlador SDN receba decisões e envie ações para serem sincronizadas. Note que todas as ações decididas pela *VNF-Consensus* são executadas sem a participação direta de qualquer controlador. A vantagem da estratégia proposta é que a *VNF-Consensus* mantém a consistência do plano de controle distribuído enquanto que o controlador está livre para realizar as suas atividades usuais. Como consequência, a solução não aumenta a carga de trabalho nem os recursos computacionais dos controladores, uma vez que as instâncias da *VNF-Consensus* são executadas em um outro *host*. A *VNF-Consensus* consegue manter o plano de controle consistente independente do número de controladores. Destaca-se também que a solução pode ser implementada sem nenhuma alteração no protocolo SDN ou nos *switches*.

Resultados experimentais obtidos com simulação são apresentados e mostram o ganho de desempenho que pode ser obtido ao utilizar a *VNF-Consensus*. Em particular, é apresentado que é possível sincronizar o plano de controle sem aumentar a carga de trabalho dos controladores, o que gera ganhos significativos tanto na vazão quanto na latência.

O restante deste capítulo está organizado da seguinte forma. A Seção 5.1 apresenta o problema de sincronização do plano de controle em redes SDN. A Seção 5.2 apresenta uma visão geral das soluções que fazem a sincronização do plano de controle em redes SDN. A Seção 5.3 descreve a solução proposta para sincronização dos controladores SDN. Os resultados experimentais parciais são descritos na Seção 5.4. Por fim, a Seção 5.5 apresenta os resultados experimentais.

5.1 Justificativa da Proposta

As redes SDN são flexíveis, programáveis e mais fáceis de serem gerenciadas do que as redes tradicionais [Sasaki et al. 2016]. Nessas redes, o objetivo principal é separar o plano de

controle do plano de dados, centralizando o controle lógico da rede e permitindo uma integração genérica entre os dispositivos da camada de controle e da camada de enlace. Em um primeiro momento, o plano de dados não possui informações de como os pacotes devem ser encaminhados entre os *switches* e roteadores. Dessa forma, estas redes contam com controladores SDN responsáveis por configurar o encaminhamento de pacotes no plano de dados. Existem várias maneiras de realizar a comunicação entre o plano de dados e o plano de controle, sendo os mais comuns o protocolo OpenFlow [McKeown et al. 2008a] e a linguagem P4 [Bosshart et al. 2014]. Em especial, este capítulo utiliza o protocolo OpenFlow.

O protocolo OpenFlow permite a configuração das tabelas de fluxo dos *switches*. Cada *switch* possui uma tabela de fluxo – uma tabela que especifica, para cada tipo de fluxo, quais ações devem ser executadas. O tipo de fluxo pode ser determinado, por exemplo, pela sua origem e pelo seu destino. Caso o pacote recebido não tenha uma entrada na tabela de fluxo, ele deverá ser encaminhado à camada de controle por meio do protocolo OpenFlow. O controlador, presente na camada de controle, irá enviar o pacote para a camada de aplicação, que irá determinar quais ações executar para aquele tipo de fluxo. Essas ações são salvas na tabela de fluxo do *switch*, fazendo com que os próximos pacotes recebidos daquele tipo não sejam encaminhados novamente ao plano de controle.

Em geral, o plano de controle em redes SDN é centralizado, assim o estado da rede é mantido através de um único controlador SDN [Ho et al. 2016]. Apesar de uma abordagem centralizada possuir várias vantagens quanto ao gerenciamento e o controle, representa por outro lado uma vulnerabilidade, uma vez que o controlador é um único ponto de falha. Além disso, a disponibilidade, escalabilidade e desempenho de um plano de controle centralizado podem estar abaixo do nível exigido por uma rede em produção [Canini et al. 2015].

Existem diversas propostas para distribuir o plano de controle em redes SDN [Schiff et al. 2016]. A fim de aprimorar a disponibilidade do plano de controle é necessário empregar técnicas de redundância [Koponen et al. 2010, Berde et al. 2014]. Um dos maiores desafios é a necessidade de garantir a consistência de operações na rede, já que ações realizadas no plano de controle por múltiplos controladores SDN precisam ser sincronizadas, e essa não é uma tarefa trivial [Schiff et al. 2016]. Considere, por exemplo, a instalação de novas regras de encaminhamento de pacotes em diversos *switches* SDN. Se regras conflitantes são instaladas, resultados incorretos podem ocorrer na rede, incluindo, por exemplo, a criação de laços ou rotas que incorretamente contornam serviços importantes (*e.g.*, *firewalls*, *Deep Packet Inspection*).

Canini et al. [Canini et al. 2015] afirmam que manter um plano de controle distribuído consistente é um dos maiores problemas em aberto em SDN. Além disso, o problema deve ser resolvido sem impacto no desempenho da rede e preservando as operações corretas do plano de dados. Algumas das soluções para desenvolver um plano de controle SDN distribuído e robusto são baseadas na sincronização de múltiplos controladores SDN no próprio plano de controle. Neste caso, os próprios controladores realizam as tarefas de sincronização. Note que a maior desvantagem desta abordagem é o custo computacional adicional causado nos controladores SDN [Koponen et al. 2010, Canini et al. 2015, Ho et al. 2016]. Por outro lado, existem outras soluções que minimizam a carga de trabalho dos controladores utilizando dispositivos de rede (*e.g.*, *switches*) para sincronizar as ações da rede SDN no plano de dados [Schiff et al. 2016, Dang et al. 2015]. Em geral, estas soluções possuem desvantagens já que necessitam mudanças no protocolo SDN (*e.g.*, OpenFlow) [McKeown et al. 2008b] ou nos próprios *switches*.

Neste capítulo é proposta uma solução para manter a consistência de um plano de controle SDN distribuído. São utilizados conceitos de NFV para definir uma estratégia particularmente eficiente em termos da carga imposta sobre os controladores. A estratégia proposta é projetada

para manter a consistência do plano de controle distribuído que consiste de múltiplos controladores SDN sem aumentar o número de tarefas que um controlador deve executar. A solução baseia-se em uma VNF denominada de *VNF-Consensus*, que implementa o algoritmo clássico de consenso Paxos [Lamport 1998] para garantir a consistência do plano de controle distribuído através da sincronização das ações realizadas pelos múltiplos controladores SDN.

5.2 Sincronização de Dados em Redes SDN

Esta seção descreve as estratégias existentes para a sincronização de múltiplos controladores SDN. A principal diferença da solução proposta é o suporte à sincronização consistente do plano de controle distribuído através de uma função virtualizada de rede.

Um *framework* de sincronização para o plano de controle baseado em transações atômicas é proposto em [Schiff et al. 2016]. Os *switches* são sincronizados a fim de garantir a consistência das operações da rede. Em particular, os autores propõem primitivas de sincronização que possibilitam um controlador executar múltiplos comandos de configuração do plano de dados como uma transação atômica. A construção da sincronização é implementada *in–band*, ou seja, no plano de dados nos *switches*. Os autores argumentam que os protocolos convencionais que sincronizam os controladores de maneira *out–of–band* (*i.e.*, fora dos *switches*) possuem um custo computacional elevado. Em contraste, soluções *in–band* permitem que os controladores resolvam problemas fundamentais de consenso através da troca de um número mínimo de mensagens. As primitivas de sincronização são implementadas utilizando a abordagem *compare–and–set* (CAS). Utilizando CAS, o *framework* garante transações atômicas que evitam cenários inconsistentes. Em outras palavras, o *framework* modifica o comando OpenFlow *FlowMod* para evitar que os controladores instalem regras inconsistentes nos *switches*. O comando *FlowMod* utiliza marcadores para definir quais regras são adicionadas ou removidas. Os autores também apresentam uma implementação simples como prova de conceito para mostrar a eficiência da solução proposta. O *framework* pode ser implementado sem nenhum hardware ou protocolo especial, entretanto demanda modificações no protocolo OpenFlow.

Em [Dang et al. 2015] é explorada a implementação do algoritmo de consenso Paxos no próprio *switch* SDN através de duas possíveis abordagens. A primeira implementa a lógica completa do Paxos (*i.e.*, sem a utilização de nenhuma otimização do protocolo). A segunda implementa um protocolo otimista denominado de *NetPaxos*, que não necessita do coordenador do Paxos. Os autores sugerem que implementar o algoritmo de consenso nos *switches* reduz a complexidade da aplicação, reduzindo também a latência das mensagens e aumenta a vazão das transações. Entretanto, ao utilizar as estratégias propostas é necessário realizar mudanças no *firmware* dos *switches*.

A fim de alcançar um estado consistente entre os controladores SDN, uma versão do algoritmo de consenso Paxos denominada *Fast Paxos-based Consensus* (FPC) é proposta em [Ho et al. 2016]. O algoritmo FPC é implementado nos próprios controladores SDN. O algoritmo Paxos foi escolhido para garantir consistência forte. De acordo com os autores, FPC é menos complexo para implementar do que o Paxos original. Comparado com o Paxos, FPC não possui um coordenador predefinido. Qualquer processo FPC pode se tornar líder (denominado de *chairman*). Uma vez que todos os processos FPC (controladores) realizaram uma atualização, o *chairman* muda o seu papel e finaliza a rodada de consenso. Os autores comparam o FPC e o algoritmo de consenso Raft [Ongaro and Ousterhout 2014] e concluem que o FPC é melhor em termos de desempenho.

Onix [Koponen et al. 2010] é uma plataforma que permite que o plano de controle SDN seja implementado como um sistema distribuído. O plano de controle mantido pelo Onix

oferece uma visão global da rede. O Onix é executado em um *cluster* de servidores físicos. Um controlador armazena o estado da rede em uma estrutura denominada NIB (*Network Information Base*). A NIB é o elemento principal do modelo de controle do Onix e é utilizada como base para o modelo distribuído. O Onix realiza a replicação e distribuição das NIBs entre as diversas instâncias de rede. Onix utiliza o protocolo Zookeeper [Hunt et al. 2010a] para a sincronização entre múltiplas instâncias. Ao utilizar o protocolo Zookeeper, as informações atualizadas da rede são disseminadas para todas as instâncias a fim de garantir a consistência.

Outra proposta segue o caminho de definir um modelo formal para descrever a comunicação entre o plano de dados e o plano de controle distribuído [Canini et al. 2015]. Os autores descrevem os problemas de consistência que podem ocorrer quando políticas de rede são atualizadas em um ou mais *switches*. Informalmente, eles garantem que todo pacote que trafega na rede deve ser processado por exatamente uma única política de rede global, mesmo quando as políticas da rede são atualizadas. Um algoritmo denominado *FixTag* permite que os controladores apliquem diretamente suas atualizações no plano de dados e resolvam os conflitos. O trabalho também apresenta um protocolo baseado em uma máquina de estados a fim de implementar uma atualização de políticas com ordenação total. O plano de dados é visualizado como uma estrutura de memória compartilhada. Além disso, os controladores são visualizados como processos que podem modificar as regras de encaminhamento aplicadas nos pacotes em cada *switch*. Dado um limite superior na latência da rede e assumindo no máximo f processos/controladores falhos, o protocolo funciona corretamente.

OpenDaylight[1] (ODL) é uma plataforma SDN. Entre as suas diversas funcionalidades, o plano de controle ODL permite a sincronização de múltiplos controladores SDN. ODL define uma arquitetura distribuída denominada de ODL *Clustering*, que fornece todos os serviços necessários para a integração dos controladores e suas aplicações. Cada controlador armazena dados localmente e a replicação de dados é baseada em cache distribuída no modelo utilizado no banco de dados distribuído Infinispan[2]. A comunicação dos controladores e as notificações entre os controladores na arquitetura ODL *Clustering* é feita através do protocolo Akka. O algoritmo de consenso Raft é utilizado e garante a consistência mesmo após atualizações realizadas por múltiplos controladores. Note que a sincronização exige a execução de diversos algoritmos embutidos nos controladores. Como resultado, os controladores possuem uma carga de trabalho maior para gerenciar a consistência no plano de controle.

A Tabela 5.1 mostra as estratégias de sincronização adotadas pelos trabalhos relacionados apresentados nesta seção. A principal diferença entre a estratégia proposta e as outras estratégias é o local onde as tarefas de sincronização são executadas. Nossa solução não impõe uma sobrecarga significativa nos controladores SDN nem nenhuma modificação nos *switches*. A próxima seção descreve em detalhes como a solução proposta funciona.

5.3 Uma VNF para Manter o Plano de Controle Consistente

Nesta seção é definido o problema de construir um plano de controle SDN consistente. Em seguida, é descrita a arquitetura da solução baseada em uma função virtualizada de rede para resolver este problema.

[1]https://www.opendaylight.org/
[2]http://infinispan.org/

Tabela 5.1: Comparação entre as estratégias de sincronização.

Trabalho Relacionado	**Algoritmo de Sincronização**	**Execução do Algoritmo**
[Schiff et al. 2016]	Transações atômicas utilizando *compare-and-set* (CAS)	*Switch*
[Dang et al. 2015]	*NetPaxos*	*Switch*
[Ho et al. 2016]	*Fast Paxos-based consensus* (FPC)	Controlador
[Koponen et al. 2010]	Zookeeper	Controlador
[Canini et al. 2015]	Algoritmo de serialização de políticas baseado em máquina de estados	Controlador
ODL Platform	*Raft*	Controlador

5.3.1 Descrição do Problema

Um plano de controle SDN distribuído consistente garante que uma sequência de operações realizadas por um conjunto de controles são executadas na mesma ordem. Controladores executam operações de rede de maneira concorrente e precisam sincronizar estas operações. As operações de rede mudam o estado da rede, por exemplo, através da instalação de regras que estabelecem novas rotas. Manter um plano de controle distribuído consistente é essencial para evitar cenários incorretos resultantes de configurações inconsistentes.

Tradicionalmente, um *switch* SDN comunica-se com um controlador e o controlador gerencia o *switch*. As decisões tomadas no plano de controle geralmente implicam em mudanças no plano de dados. O controlador adiciona uma entrada de fluxo na tabela de fluxo do *switch* em resposta a uma mensagem do *switch* [McKeown et al. 2008b]. Existem dois tipos de estratégias de comunicação descritas para interações entre os componentes em uma rede SDN (Figura 5.1) [Tianzhu et al. 2016]: *Switch-to-Controller* e *Controller-to-Controller*, ambas detalhadas a seguir.

As comunicações *Switch-to-Controller* suportam a interação entre o *switch* e o controlador através de um protocolo SDN. Este tipo de interação ocorre por exemplo, quando um *switch* OpenFlow encaminha uma mensagem do tipo *packet_in* para o controlador quando não existe nenhuma entrada para este tipo de fluxo na tabela de fluxos do *switch*. Em resposta, o controlador retorna uma mensagem do tipo *FlowMod* que permite ou recusa a instalação de uma nova entrada de fluxo. Mensagens *FlowMod* são utilizadas para gerenciar a tabela de fluxo de cada *switch*. Considerando múltiplos controladores, uma decisão para adicionar uma nova entrada de fluxo precisa ser sincronizada entre todos os controladores SDN para evitar problemas como rotas inconsistentes que podem causar perda de pacotes ou resultar em laços indesejados. Note que também é necessário que os *switches* instalem as mensagens *FlowMod* de maneira a evitar cenários inconsistentes. Entretanto, pode ser complexo e computacionalmente custoso atualizar o plano de dados de modo que a consistência seja garantida. Além disso, é preciso lidar com diversos problemas, como o fato das redes reais não serem síncronas, no sentido de que o limite superior no atraso de transmissões de mensagens não pode ser sempre previsto [Zhou et al. 2014]. Neste capítulo é proposta uma estratégia para construir um plano de controle consistente que garante a sincronização do plano de dados.

Já as comunicações *Controller-to-Controller* permitem a interação direta entre os controladores. Para atingir consistência entre os controladores, é necessário que todos tenham a mesma visão do estado da rede. A consistência pode ser forte ou eventual [Tianzhu et al. 2016]. Apesar de ambas garantirem a consistência em operações de escrita, a consistência forte implica

que toda operação de leitura a partir de diferentes controladores sempre levam a um mesmo resultado. Por outro lado, a consistência eventual significa que uma leitura pode levar a diferentes resultados por um curto período de tempo. Na arquitetura proposta é utilizado o algoritmo de consenso Paxos, que fornece uma consistência forte [Van Renesse and Altinbuken 2015, Rao et al. 2011].

5.3.2 A Função Virtualizada de Rede *VNF-Consensus*

A *VNF-Consensus* implementa um algoritmo de consenso para manter a consistência no plano de controle SDN. Em particular, a VNF proposta é baseada no algoritmo de consenso Paxos, descrito em detalhes no Capítulo 2. Relembrando, o Paxos é um algoritmo de consenso tolerante a falhas projetado para replicação de máquina de estados [Lamport 1998]. Informalmente, o consenso permite que um conjunto de processos proponham valores iniciais diferentes e entrem em acordo sobre um valor final. Um algoritmo de consenso deve satisfazer as propriedades de progressão (*liveness*) e propriedades de segurança (*safety*): integridade, acordo e validade. A progressão garante que em algum momento o consenso será atingido. O algoritmo Paxos garante a segurança em um sistema assíncrono sujeito a falhas e o progresso sob suposições de assincronia fraca e considerando que a maioria dos processos estão corretos. Além disso, o modelo de falha é parada com recuperação (*crash-recovery*).

Na estratégia proposta, o consenso é executado para cada *packet_in* recebido pelo controlador (*e.g.*, regras OpenFlow). Depois que uma decisão é alcançada pela *VNF-Consensus*, cada controlador recebe o valor decidido e instala a regra correspondente no plano de dados. Isso pode ser feito se considerarmos que cada controlador SDN se comporta como um *learner* do algoritmo Paxos. Como o Paxos fornece consistência forte, todos os controladores em algum momento terão o mesmo conjunto de dados (*i.e.*, as mesmas entradas da tabela de fluxo). Ou seja, a *VNF-Consensus* pode garantir a consistência do plano de controle distribuído a partir do ponto de vista de qualquer controlador da rede SDN.

A fim de sincronizar o estado da rede, cada controlador é configurado como um *proposer* e um *learner*, para enviar e receber valores da *VNF-Consensus*, enquanto que a *VNF-Consensus* é configurada com um *acceptor*. Um controlador SDN recebe decisões de uma instância da *VNF-Consensus* e envia ações para serem sincronizadas. Note que todas as decisões tomadas no contexto da VNF são executadas *fora* do controlador, *i.e.*, não consomem recursos computacionais do controlador. Após uma decisão, o controlador executa a ação recebida pela *VNF-Consensus*.

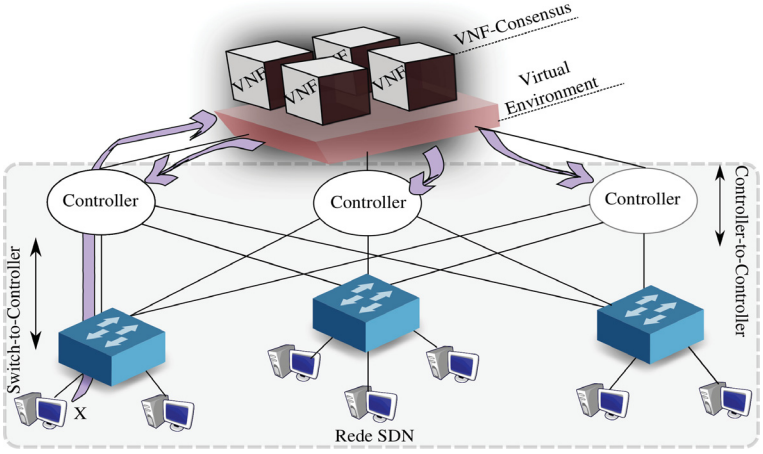


Figura 5.1: Uma rede SDN executando a *VNF-Consensus*.

A Figura 5.1 mostra a interação entre os controladores com a *VNF-Consensus*. Quando uma nova regra na rede precisa ser sincronizada, o respectivo controlador encaminha esta regra para a *VNF-Consensus*. Após fazer este encaminhamento, o controlador pode continuar atendendo outras requisições da rede SDN. Ou seja, não é necessário que o controlador aguarde a decisão do consenso. Depois que uma decisão é tomada, todos os controladores recebem o resultado final e atualizam seu estado local. Embora a figura mostre diversas instâncias da *VNF-Consensus*, não é necessário que cada controlador possua uma instância associada da *VNF-Consensus*. Uma única instância pode ser utilizada por todos os controladores, ou mais instâncias de acordo com os requisitos de escalabilidade.

Considere qualquer *host* na Figura 5.1. Assuma que este *host* comece a enviar um fluxo de pacotes que chegue em um *switch* OpenFlow. O *switch* possui uma tabela de fluxos e todos os pacotes recebidos são comparados com as entradas desta tabela de fluxos. Se uma entrada correspondente não é encontrada, uma mensagem *packet_in* é enviada para o controlador. Na estratégia proposta, após um controlador receber esta mensagem do *switch*, a regra correspondente é encaminhada para a *VNF-Consensus* antes de ser instalada. A instância da *VNF-Consensus* recebe a regra, executa o algoritmo de consenso e retorna a decisão aos controladores, que por sua vez instalam a regra decidida.

5.4 Avaliação Experimental

Nesta seção são apresentados os resultados de vários experimentos executados com simulação para avaliar a *VNF-Consensus*. O experimento foi executado em um *host* com processador AMD FX-4300 3.8 GHz de 4 núcleos e utilizando o sistema operacional *Ubuntu* 16.04. Para simular a rede SDN foram utilizados controladores Ryu[3] e *switches* virtuais OpenVSwitch[4]. Cbench[5] é uma ferramenta de *benchmarking* utilizada para testar a maior taxa de *packet_in* que um controlador suporta. Em todos os experimentos Cbench é configurada para o modo *throughput*. A biblioteca utilizada que implementa o Paxos é a libPaxos[6]. A *VNF-Consensus* utiliza uma interface REST para comunicar com os controladores.

Tipicamente a *VNF-Consensus* é inicializada com três instâncias, de forma a tolerar duas falhas. Cada instância da *VNF-Consensus* é executada em um *container* da plataforma Docker[7]. Note que cada regra implica em executar uma instância individual do Paxos. Como resultado, as VNFs são totalmente independentes e isoladas. Os *containers* permitem a execução isolada de aplicações em um determinado *host*. Destaca-se que *containers* são uma estratégia de virtualização que apresenta uso de recursos físicos significativamente menor, comparado a máquinas virtuais tradicionais. Cada controlador Ryu é executado em um *container* individual.

Os resultados são separados em três conjuntos de experimentos. O primeiro conjunto foi executado para avaliar o custo de sincronização das ações no plano de controle SDN com base em três métricas: (1) uso de CPU; (2) número de fluxos por segundo suportado pelo controlador enquanto o número de *switches* aumenta; (3) o tempo de instalação de um conjunto de regras pelo controlador. O segundo conjunto de experimentos têm como objetivo avaliar a vazão do consenso, *i.e.*, o número de decisões do consenso por instante de tempo. Além da vazão, é feita a comparação com uma implementação alternativa em que os próprios controladores são responsáveis por executar o consenso e manter o plano de controle consistente. Por fim, o último

[3]https://osrg.github.io/ryu/
[4]https://www.openvswitch.org/
[5]https://github.com/mininet/oflops/tree/master/cbench
[6]https://bitbucket.org/sciascid/libpaxos
[7]https://www.docker.com/

conjunto de experimentos foi executado para avaliar o desempenho da solução proposta conforme o número de controladores aumenta.

5.4.1 Custo de Sincronização do Plano de Controle

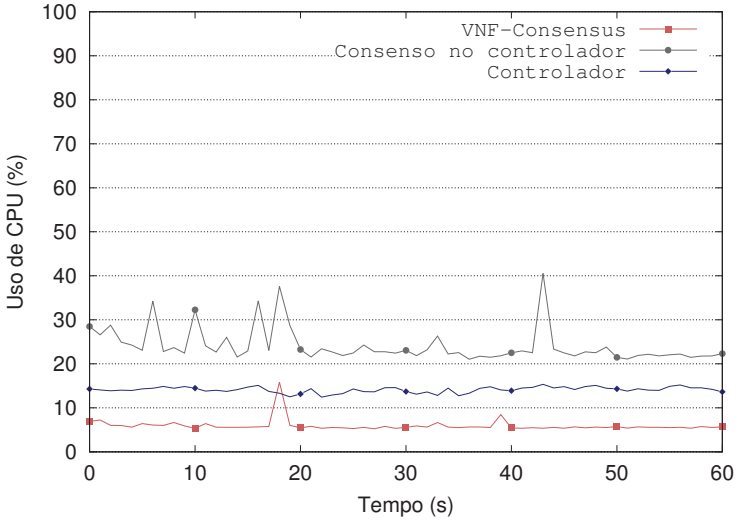
Enquanto a quantidade de tarefas executadas pelo controlador aumenta, seus recursos computacionais podem ser sobrecarregados a um ponto em que eles não conseguem executar suas atividades básicas, resultando em um impacto na disponibilidade do controlador. A principal vantagem da solução proposta é manter o plano de controle consistente sem aumentar a carga nos controladores. No primeiro conjunto de experimentos, é comparada a solução *VNF-Consensus* com a solução em que os próprios controladores estão encarregados de manter a consistência do plano de controle distribuído. A Figura 5.2 mostra os resultados para três métricas: (1) uso de CPU; (2) número de fluxos suportado pelo controlador; (3) tempo que o controlador utiliza para instalar um conjunto de regras nos *switches*.

A Figura 5.2(a) apresenta três cenários: (1) o uso de CPU do controlador enquanto executa suas operações regulares enquanto que a *VNF-Consensus* é responsável por manter a consistência do plano de controle (curva *Controlador*). Neste caso o controlador apenas encaminha as regras para a *VNF-Consensus*. É importante notar que o controlador não permanece bloqueado aguardando respostas da *VNF-Consensus*; (2) o uso de CPU do controlador enquanto executa o Paxos além das suas atividades regulares (curva *Consenso no Controlador*); (3) uso de CPU da *VNF-Consensus* (curva *VNF-Consensus*). Para cada métrica três amostras foram coletadas, cada uma em um intervalo de 60 segundos. A topologia da rede foi criada com três controladores e três instâncias da *VNF-Consensus*. Cada controlador gerencia um *switch*.

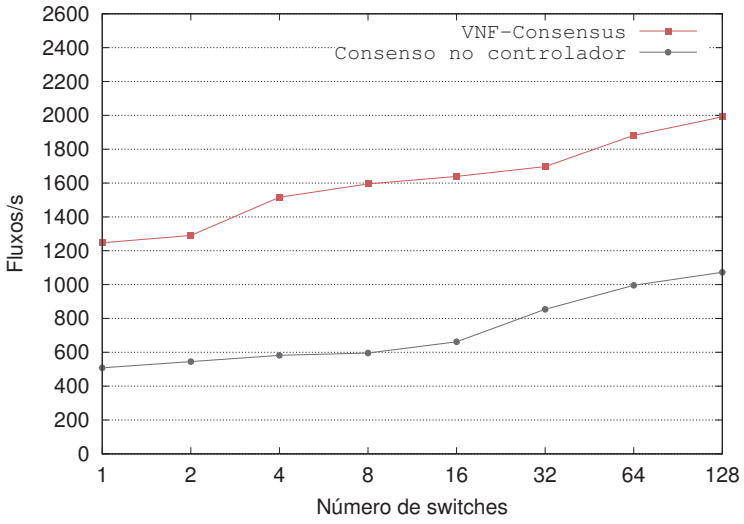
Note que, quando o controlador executa o algoritmo de consenso, o uso de CPU é em média, 27.1% (curva *Consenso no Controlador*). Por outro lado, ao utilizar a *VNF-Consensus* a média do uso de CPU diminui para 14.3%. O uso de CPU da *VNF-Consensus* é em média 16%. Este experimento mostra claramente a vantagem de desacoplar a execução do consenso do controlador. Como consequência, a disponibilidade do controlador aumenta. A razão é que o custo de sincronização é realocado para uma entidade externa, *i.e.,* uma função virtualizada de rede.

A Figura 5.2(b) mostra outra vantagem da *VNF-Consensus*. O principal objetivo é medir o impacto no controlador quando ele é responsável por gerenciar tanto a rede SDN (*e.g.*, instalando regras nos *switches*) quanto o plano de controle consistente. A rede consiste de três controladores e três VNFs enquanto que o número de *switches* aumenta de 1 até 128. Cada experimento foi executado durante 1 minuto e foi repetido a cada vez que o número de *switches* aumentou. Enquanto o número de *switches* aumenta, um número maior de fluxos é criado. Por exemplo, considere que 8 *switches* geram 600 *packet_in* por segundo. Isso gera um total de 4800 fluxos por segundo que precisam ser gerenciados pelo controlador. O eixo *y* mostra o número de fluxos por segundo por *switch*. Como mostrado na Figura 5.2(b), quando a *VNF-Consensus* realiza a sincronização, o controlador pode gerenciar um número maior de requisições. Isso acontece devido ao controlador possuir uma carga de trabalho menor em comparação quando ele deve realizar tarefas de sincronização do plano de controle. A diferença é significativa e de aproximadamente 53%.

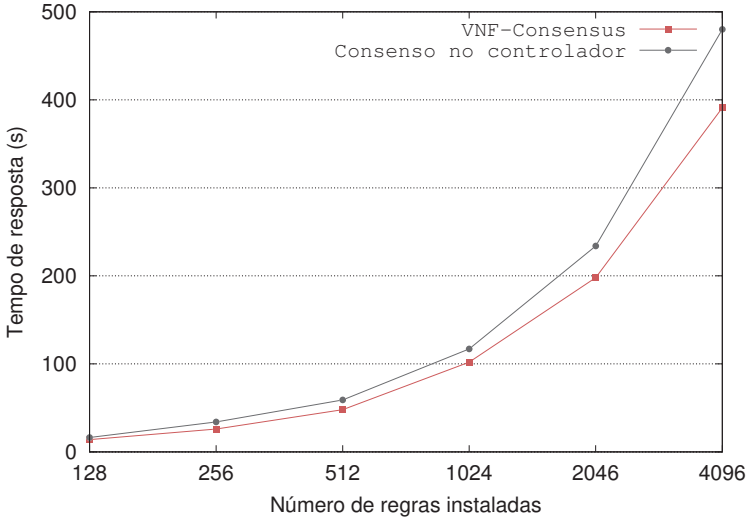
No terceiro experimento mostrado na Figura 5.2(c), um *script* foi criado para gerar requisições REST contínuas com o objetivo de sobrecarregar o controlador. Foi então medido o tempo necessário para instalar um conjunto de regras considerando todo o caminho percorrido (*i.e.*, *Host* –> *Switch* –> Controlador –> *Switch* –> *Host*). Neste experimento foram utilizados três controladores e três VNFs. Na Figura 5.2(c) enquanto o número de regras são gradualmente



(a) Utilização de CPU.

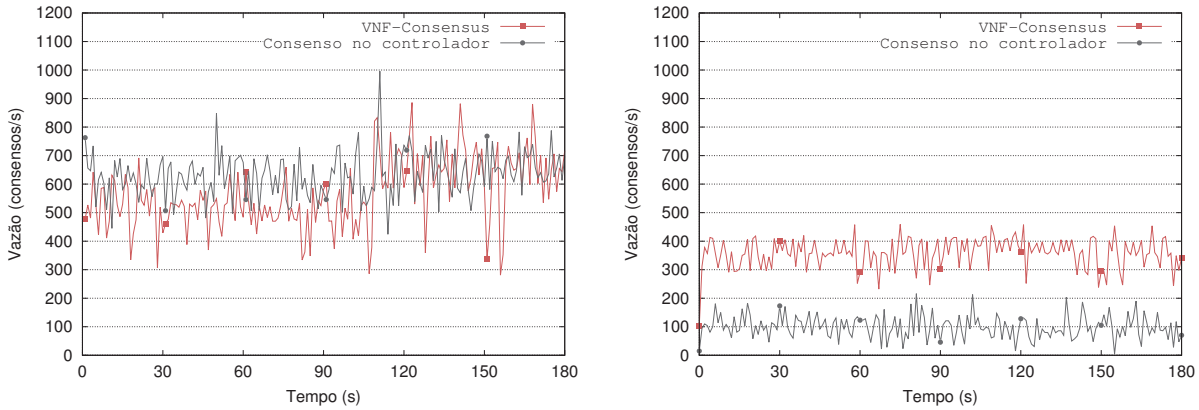


(b) Fluxos por segundo suportado pelo controlador.



(c) Tempo para um controlador instalar um conjunto de regras no *switch*.

Figura 5.2: Sincronização do plano de controle: avaliação do desempenho.



(a) Medida de vazão sem considerar tarefas paralelas nos controladores.

(b) Medida de vazão considerando tarefas paralelas nos controladores.

Figura 5.3: Comparando a vazão das soluções.

instaladas, o tempo de resposta também aumenta. Entretanto, a curva *VNF-Consensus* mostra uma redução de até 18.5% no tempo de resposta. Isso significa que a estratégia proposta supera outras que executam a sincronização nos próprios controladores.

5.4.2 Vazão da *VNF-Consensus*

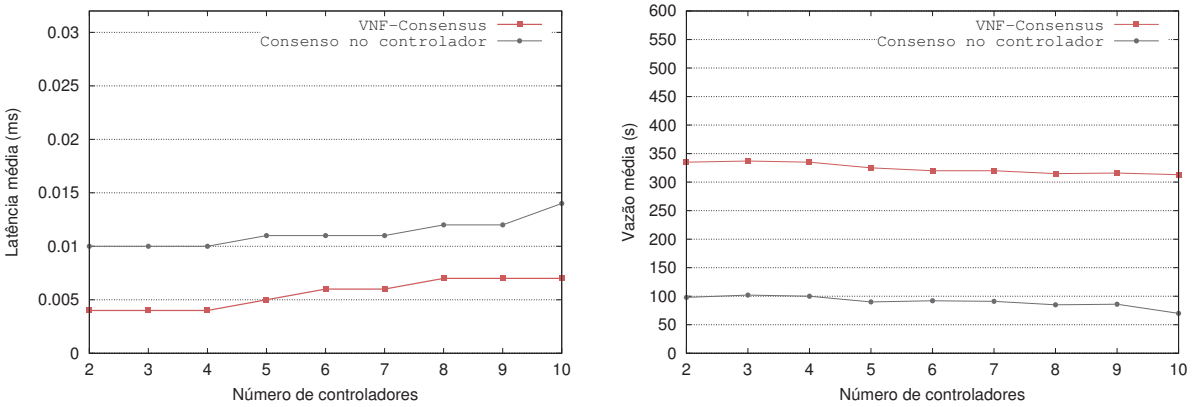
Neste conjunto de experimentos é medida a vazão do consenso quando o Paxos é executado como uma VNF ou nos próprios controladores. A topologia deste experimento também consiste de três controladores e três VNFs e cada controlador gerencia um único *switch*. O eixo y mostra o número de consensos completados por segundo durante três minutos (eixo x). Cada experimento foi executado três vezes.

Na Figura 5.3(a), requisições de atualização são continuamente submetidas ao controlador. Entretanto, o controlador não está executando nenhuma outra tarefa em paralelo, *i.e.*, é dedicado a executar o consenso a cada requisição de atualização. A curva *Consenso no Controlador* mostra a vazão do consenso neste cenário. A curva *VNF-Consensus* mostra a vazão quando a VNF está responsável por realizar a sincronização. O resultado confirma as expectativas, uma vez que as instancias das VNFs estão em um *container* separado, ou seja, fora dos controladores. Por consequência, a vazão da *VNF-Consensus* é menor devido aos passos adicionais de comunicação entre o controlador e a VNF. Paxos no controlador possui uma vazão 11.4% maior do que da solução em VNF.

Em contraste, no experimento mostrado na Figura 5.3(b), o controlador além de gerenciar o fluxo de dados, executa tarefas de sincronização. Em outras palavras, o controlador está executando múltiplas tarefas em paralelo e portanto possui uma carga de trabalho maior. Como consequência, note que a vazão de ambas as soluções diminui drasticamente. Entretanto, neste cenário a vazão da *VNF-Consensus* é 3,6 vezes maior do que a alternativa de executar todas as tarefas no controlador. Isso significa que a solução baseada em VNF fornece ganhos significativos em eficiência quando consideramos cenários com alta carga de trabalho.

5.4.3 Aumentando a Quantidade de Controladores

O último experimento investiga a latência e a vazão do consenso enquanto o número de controladores SDN varia. Inicialmente, a topologia da rede consiste de um *switch* por controlador



(a) Latência média do Paxos variando o número de controladores.

(b) Vazão do Paxos variando o número de controladores.

Figura 5.4: Comparando a escalabilidade.

e três instâncias da *VNF-Consensus*. No experimento o número de controladores continua aumentando até 10. O número de instâncias de VNFs permanece constante em três, ao passo que quando o Paxos é executado no controlador, o número de instância é proporcional ao número de controladores. É importante notar que nesta alternativa, os controladores têm múltiplas tarefas em paralelo.

A Figura 5.4(a) mostra a latência do Paxos quando o algoritmo está executando nos controladores e como uma VNF. Este experimento mostra que a *VNF-Consensus* possui a menor latência. Em especial, uma instância de Paxos é executada em média em 0,005ms em uma VNF, ao passo que no controlador ela executa em média em 0,011ms. Portanto, a *VNF-Consensus* reduz a latência em cerca de 54%. Além disso, note que o número de instâncias de *VNF-Consensus* é configurável e não depende do número de controladores. Em outras palavras, a latência do Paxos executando nos controladores aumenta de maneira aproximadamente linear conforme aumenta o número de controladores.

A Figura 5.4(b) mostra a comparação entre a *VNF-Consensus* e o Paxos no controlador em termos do número de execuções do consenso executada por segundo enquanto o número de controladores aumenta até 10. Como pode ser observado, a solução baseada em VNF apresenta uma vazão maior do que o Paxos executado no controlador, tendo em média, uma vazão 2,8 vezes maior. Note que o número de instâncias de VNF permanecem constantes em três. Portanto, mesmo quando a quantidade de controladores aumenta, por exemplo de 3 a 10 controladores, a *VNF-Consensus* ainda apresenta a melhor vazão.

A partir dos resultados apresentados nesta seção podemos concluir que a *VNF-Consensus* apresenta melhor desempenho e é mais escalável do que a solução alternativa de executar o consenso nos próprios controladores.

5.5 Conclusões Parciais

Neste capítulo foi proposta a *VNF-Consensus*, uma função virtualizada de rede que implementa o algoritmo Paxos para garantir a consistência de um plano de controle distribuído em redes SDN através da sincronização de todas as ações executadas pelos controladores. Utilizando esta abordagem, os controladores estão livres de executar qualquer tipo de ação relacionada a sincronização e podem executar suas atividades regulares. É descrito também como o Paxos pode

ser utilizado em uma função virtualizada de rede para garantir a consistência de operações de rede em uma rede SDN. Resultados experimentais são apresentados, comparando a *VNF-Consensus* com uma solução alternativa onde os próprios controladores são responsáveis por manter a consistência das operações.

Os resultados são classificados em três conjuntos. Em todos os eles, a *VNF-Consensus* claramente mostra sua eficiência em termos de recursos computacionais, vazão do número de execuções de consenso por unidade de tempo e escalabilidade. Em geral os resultados comparam o Paxos executando em controladores SDN e como uma VNF e confirmam que a solução proposta mostra ganhos significativos de desempenho.

A *VNF-Consensus* fornece uma alternativa de projetar um plano de controle distribuído consistente em redes SDN através da utilização de funções virtualizadas de rede, sem aumentar as tarefas que o controlador executa.

6 *AnyBone*: Um *Backbone* Virtual com Serviços de Difusão Confiável e Ordenada de Mensagens

Este capítulo apresenta o *AnyBone*: um *backbone* virtual baseado em NFV que oferece serviços de difusão de mensagens implementados na própria rede. Os serviços de difusão são frequentemente utilizados para a construção de aplicações distribuídas tolerantes a falhas. Esta solução oferece uma primitiva de difusão confiável e três primitivas de difusão ordenada de mensagens: FIFO, causal e atômica. A difusão confiável garante a entrega das mensagens transmitidas em uma rede SDN por todos os processos corretos. As estratégias de difusão ordenada garantem que as mensagens são entregues todas na mesma ordem (atômica), de acordo com a ordem em que foram enviadas pela origem (FIFO) ou obedecendo a ordem causal. O *AnyBone* garante a ordem das mensagens através de um sequenciador, também implementado como uma VNF, denominada de *VNF-Sequencer*. A estratégia proposta foi implementada e resultados experimentais são apresentados.

O restante deste capítulo está organizado da seguinte forma. O contexto detalhado da presente proposta é apresentada na Seção 6.1. A Seção 6.2 apresenta plataformas e protocolos que implementam a difusão atômica. Na Seção 6.3 são apresentados os detalhes de funcionamento dos algoritmos de difusão confiável disponibilizados pelo *AnyBone*, bem como a arquitetura e a lógica de implementação. A Seção 6.4 mostra os experimentos realizados que comprovam e permitem demonstrar as funcionalidades implementadas pelo *AnyBone*. Por fim, as conclusões parciais são apresentadas na Seção 6.5.

6.1 Justificativa da Proposta

Uma abstração importante para o desenvolvimento de aplicações distribuídas e tolerante a falhas é a difusão confiável (*reliable broadcast*). Informalmente, a difusão confiável garante que as mensagens enviadas para um conjunto de processos seja entregue por todos os processos corretos [Défago et al. 2004]. Além disso, um processo pode exigir que todas as mensagens sejam entregues em uma determinada ordem por todos os demais processos. Neste caso, existem diversos tipos de ordens que podem ser definidos. Se todos os processos corretos devem entregar todas as mensagens exatamente na mesma ordem, a difusão é denominada atômica (*atomic broadcast*). Se a ordem for determinada a partir do processo emissor, então a difusão é denominada FIFO. Quando as mensagens devem ser entregues de acordo com a precedência causal [Lamport 1978], então a difusão é denominada causal (*causal broadcast*). Na prática, a implementação da difusão confiável e ordenada não é uma tarefa trivial. Se este serviço é oferecido na própria aplicação, a complexidade para o seu desenvolvimento é significativamente

maior [Li et al. 2016]. Outra possível alternativa é a utilização de *middlewares* específicos que executam nas máquinas dos próprios usuários.

Em [Ekwall and Schiper 2007] os autores argumentam que o desempenho dos algoritmos de difusão confiável são influenciados pelo número de passos na comunicação e pelo número de mensagens necessárias para alcançar uma decisão. No entanto, além do usuário levar em consideração aspectos de desempenho, também é necessário escolher um algoritmo que se adapte melhor às características específicas da rede onde é executado. A nova alternativa explorada retira esta preocupação do usuário, pois move a difusão confiável da máquina do usuário (onde é normalmente executada como uma aplicação ou *middleware*) para a própria rede.

Neste capítulo é proposta a implementação de um *backbone* virtual, denominado de *AnyBone*, que oferece difusão confiável utilizando Virtualização de Funções de Rede em uma rede SDN. O *AnyBone* aproveita as vantagens das tecnologias de virtualização para habilitar a rede a oferecer diversos serviços de difusão confiável e ordenada. De fato, a ordem total das mensagens é garantida através do uso de um sequenciador que também está presente dentro da própria rede, enquanto que as primitivas de difusão são executadas através de uma API (*Application Programming Interface*) acessível pela aplicação distribuída. O sequenciador é implementado como uma VNF, denominada de *VNF-Sequencer* e a API é denominada de *RBCast*, onde são oferecidos diversos tipos de difusão: difusão confiável, difusão atômica, difusão atômica FIFO e difusão atômica causal. Portanto, a *VNF-Sequencer* é responsável por gerenciar toda a troca de mensagens de forma a garantir as propriedades de comunicação definidas na aplicação distribuída.

O *AnyBone* foi implementado e resultados experimentais são descritos, incluindo a latência e vazão da difusão em diversos cenários. Por fim, os resultados demonstram que a estratégia proposta consegue garantir a difusão confiável e ordenada de mensagens de maneira simples e eficiente.

6.2 Plataformas e Protocolos de Difusão Atômica

Nesta seção são descritas plataformas e protocolos que oferecem primitivas de difusão atômica. A grande diferença entre estas estratégias e a solução proposta é que o *AnyBone* virtual utiliza funções virtualizadas de rede, sendo possível a implantação das primitivas de comunicação dentro da própria rede.

Em [Reed and Junqueira 2008] os autores propõe Zab, um protocolo de difusão atômica utilizado pelo serviço ZooKeeper [Hunt et al. 2010b]. O protocolo Zab é utilizado para manter réplicas dos dados em cada servidor ZooKeeper. O protocolo Zab considera um sequenciador fixo, denominado de líder, eleito a partir de um algoritmo de eleição de líder. Cada processo que deseja realizar uma difusão atômica deve enviar sua mensagem ao líder. Para a entrega das mensagens aos demais processos, o líder executa um algoritmo semelhante ao *two-phase commit*, onde é feita uma requisição, uma coleta de votos e posteriormente o *commit*. Para garantir a ordem FIFO, todas as comunicações utilizam conexões TCP entre todos os pares de processos do sistema.

No trabalho apresentado em [Li et al. 2016], com o objetivo de garantir a replicação consistente em *data centers*, os autores utilizam uma solução que divide as tarefas entre a rede e a aplicação. A rede garante a ordem das mensagens, enquanto que o protocolo de replicação garante a entrega das mensagens. O protocolo utilizado é denominado de NOPaxos (*Network-Ordered Paxos*). NOPaxos é executado somente quando necessário, evitando a sincronização constante entre os processos. Em outras palavras, quando ocorrem imprevistos na comunicação (*e.g.*, mensagens perdidas), o protocolo NOPaxos é executado. A ordem das mensagens é garantida pela implementação de um sequenciador localizado dentro da rede. Os autores mostram que

quando o sequenciador é implementado diretamente nos *switches*, a replicação ocorre com baixa latência e alta vazão, fornecendo uma replicação com baixo custo.

Mais antigo, o ISIS [Birman and Joseph 1987] é um sistema distribuído clássico que fornece mecanismos de tolerância a falhas. A primitiva ABCAST do ISIS fornece a difusão atômica de mensagens. Além disso, são oferecidas outras formas de ordenação, como a ordem causal e atômica causal. Os autores consideram grupos de processos e a ordem global das mensagens é garantida mesmo entre grupos sobrepostos (*i.e.*, um ou mais processos em grupos diferentes). Uma vez que garantir a ordem total das mensagens pode ser uma tarefa custosa, o sistema ISIS também fornece primitivas mais fracas de ordenação em troca de um melhor desempenho. A aplicação do usuário determina qual primitiva irá utilizar.

Em [Kaashoek and Tanenbaum 1991], os autores propõe Amoeba, um sistema operacional distribuído com primitivas de difusão de mensagens. Em especial, Amoeba fornece primitivas de difusão confiável e ordenada para grupos de processos. Pode-se assumir um número variado de grupos e um processo pode pertencer a mais de um grupo. Além disso, as mensagens disseminadas por um processo são enviadas apenas aos processos pertencentes do grupo do processo emissor. O protocolo também assume falhas de comunicação e falhas nos processos. O sistema Amoeba executa o protocolo dentro do *kernel* do sistema operacional. Dessa forma, o hardware de cada processo deve ser idêntico, executar o mesmo *kernel* e utilizar a mesma aplicação. O protocolo utiliza um sequenciador centralizado em que um dos membros do grupo recebe o papel de sequenciador.

Em geral, as plataformas e protocolos propostos executam nas máquinas dos usuários. Isso é feito através da utilização de hardware ou software especializado. Além disso, nem todas as soluções apresentadas implementam todas as primitivas importantes de difusão ordenada. A proposta deste trabalho é utilizar uma função virtualizada de rede para a implementação, na própria rede, de diversas primitivas de difusão confiável e ordenada. Dessa forma, a complexidade para construção de aplicações distribuídas é menor, uma vez que basta utilizar um serviço oferecido pela própria rede. Por fim, a estratégia baseada em NFV não requer nenhum hardware especializado, *middleware* ou modificações na aplicação para executar os diversos tipos de difusão.

6.3 *AnyBone*

Nesta seção são apresentados os algoritmos de difusão confiável implementados no *AnyBone*. A arquitetura e a lógica de implementação também são descritas.

6.3.1 Difusão Confiável de Mensagens pelo *AnyBone*

O *AnyBone* implementa uma VNF que fornece primitivas de comunicação para a transmissão de mensagens via difusão confiável e ordenada. Esta abordagem assume canais de comunicação confiáveis ponto-a-ponto e implementa vários algoritmos clássicos para garantir a entrega e a ordem das mensagens. O sistema subjacente é assíncrono, dotado de detector de falhas, que deve apresentar completude forte [Chandra and Toueg 1996]. Os algoritmos são executados de forma modular, em camadas. Por exemplo, para a entrega confiável das mensagens, é necessário sempre executar o algoritmo para a difusão confiável, ao passo que, para garantir também a ordem total das mensagens, é necessário executar também, no topo da difusão confiável, o algoritmo de difusão atômica.

O algoritmo básico de difusão confiável implementado pelo *AnyBone* [Chandra and Toueg 1996] é descrito no Algoritmo 1. Este algoritmo garante a entrega das

mensagens da seguinte forma: o processo origem que dispara a difusão, envia a mensagem *m* para todos os processos. Quando um processo recebe a mensagem *m* pela primeira vez, ele retransmite *m* para todos os processos. Cada mensagem, na difusão confiável, tem informações sobre o emissor da mensagem e o conteúdo do pacote. É importante notar que toda mensagem possui um campo, denominado $local\_seq$, composto pelo identificador do processo e o número de sequência desta mensagem. Dessa forma as mensagens podem ser identificadas unicamente e o sequenciador pode ordenar mensagens vindas concorrentemente de múltiplos processos emissores.

Algoritmo 1 Algoritmo para difusão confiável.

1: Para todo processo p_i:

2: Para executar **R_broadcast(***m***)**:
3: **envia** *m* para todos os processos (incluindo p_i)

4: **R_deliver(***m***)** ocorre da seguinte forma:
5: **if** recebe *m* pela primeira vez **and** (emissor $(m) \neq p_i$) **then**
6: **envia** *m* para todos
7: **R_deliver(***m***)**
8: **end if**

As próximas seções apresentam detalhes de como o *AnyBone* define a ordem total das mensagens transmitidas por difusão para serem entregues às aplicações, bem como a arquitetura e lógica de implementação.

6.3.2 Construindo a Ordem Total das Mensagens

Em um sistema em que é necessário manter a ordem total das mensagens, é fundamental resolver o problema de como garantir a ordenação. Na literatura existem algumas alternativas para a implementação da ordem total dos eventos em um sistema distribuído. Uma delas é a utilização de um módulo sequenciador responsável por receber todas as mensagens e encaminhá-las aos destinatários, garantindo então a ordem total das mensagens. O sequenciador pode ser fixo, móvel, baseado em privilégios, entre outros [Défago et al. 2004]. Este trabalho utiliza um sequenciador fixo para garantir a ordem das mensagens.

Nesta abordagem, a ordem total é construída da seguinte forma: para uma mensagem *m* ser transmitida por difusão, em primeiro lugar o emissor de *m* encaminha a mensagem para o sequenciador; quando *m* é recebida pelo sequenciador a mensagem obtém um valor único de sequência; *m* é transmitida por difusão para todos os destinatários; por fim, os processos receptores entregam *m* de acordo com o valor de sequência atribuído pelo sequenciador.

O sequenciador fixo pode ser implementado através de três métodos apresentados na Figura 6.1 [Défago et al. 2004]: UB (*Unicast-Broadcast*), BB (*Broadcast-Broadcast*) e UUB (*Unicast-Unicast-Broadcast*). A Figura 6.1(a) apresenta o método UB onde o processo que deseja transmitir uma mensagem por difusão (processo emissor) executa uma comunicação *unicast* (ponto-a-ponto) seguida de uma difusão executada pelo sequenciador. O sequenciador, ao executar a difusão, adiciona um número de sequência à mensagem. Ao receber a mensagem, o processo receptor entregará a mensagem para a aplicação respeitando a ordem indicada pelo sequenciador. Já no método BB apresentado na Figura 6.1(b), o processo emissor executa uma difusão para todos os destinatários, incluindo o sequenciador. O sequenciador adiciona um número de sequência na mensagem e executa uma difusão para todos os destinatários. Note

que este método gera um número maior de mensagens do que o método UB. Por outro lado, o método BB facilita a implementação de um sequenciador tolerante a falhas. Por fim, no método UUB apresentado na Figura 6.1(c), o processo emissor solicita um número de sequência para o sequenciador. O próprio processo emissor inclui o número de sequência na mensagem e a transmite para todos os processos. Este último método (UUB) é menos comum que os demais e inclui um terceiro passo que o torna menos eficiente – já que os demais resolvem o problema em apenas dois passos. O sequenciador do *AnyBone* utiliza o método BB. A implementação do sequenciador, denominada de *VNF-Sequencer*, é descrita na Seção 6.3.3.

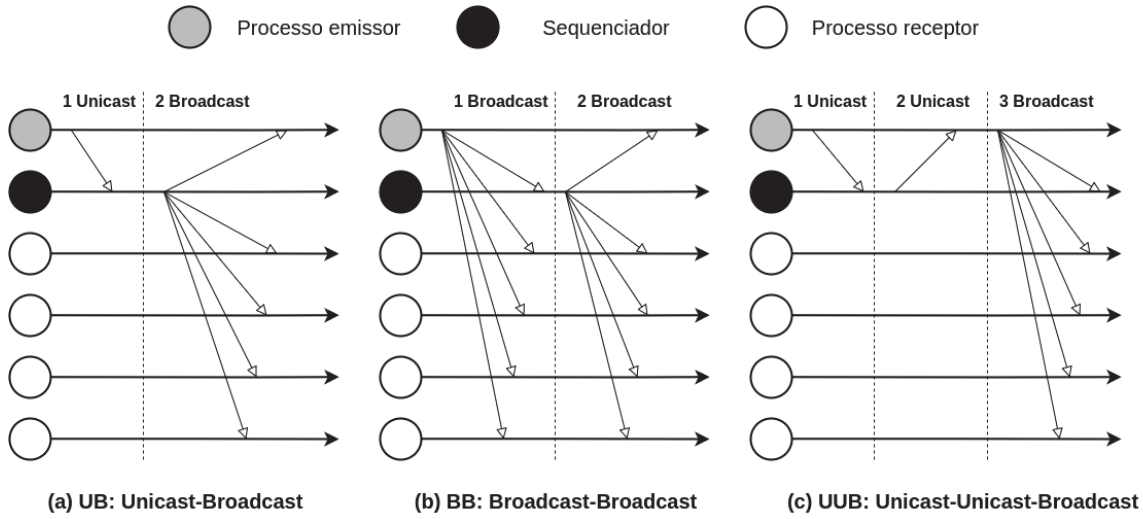


Figura 6.1: Três métodos para implementar o sequenciador fixo.

Neste trabalho, o sequenciador considera duas sequências de mensagens. A primeira respeita a ordem local de transmissão de cada processo, possibilitando por exemplo, que a ordem FIFO seja implementada. A outra é uma ordem global que possibilita garantir que a ordem de entrega será atômica. Na transmissão atômica, a ordem global é sempre a ordem em que os processos finais (receptores) consideram para a entrega das mensagens.

No Algoritmo 2 é apresentado o pseudo-código para a construção da ordem atômica entre os processos. No algoritmo existem três papéis distintos: *Sender* (processo emissor), *Sequencer* (sequenciador das mensagens) e o *Receiver* (processo receptor). O processo p_i que deseja transmitir uma mensagem m, por difusão atômica e confiável (*Sender*), inicialmente define o algoritmo a ser utilizado, por exemplo, difusão atômica, difusão atômica e FIFO ou difusão atômica e causal. A mensagem m será transmitida por difusão carregando informações como o contador local de mensagens ($local\_seq$) e informando o tipo de algoritmo que será utilizado. O contador global é inserido pelo sequenciador na mensagem e incrementado sempre após a retransmissão de uma mensagem aos processos finais. Cada processo receptor (*Receiver*) que recebe uma mensagem m, adiciona esta mensagem ao conjunto de mensagens pendentes ($pendingMsg$). Logo após é feita a verificação se existe alguma mensagem m' do processo p_i que possui o contador igual ao identificador da próxima mensagem ($nextMsg$) e também se m' ainda está em $pendingMsg$. Note que $nextMsg$ é composto pelo identificador do processo e seu respectivo contador de mensagens. Quando estas duas condições são satisfeitas, então m' e todas as outras mensagens que satisfazem as condições indicadas são entregues para a aplicação.

No caso da aplicação requerer ordenação não apenas atômica mas também FIFO, o sequenciador implementa a ordem de entrega de acordo com o Algoritmo 3. No algoritmo da difusão FIFO é necessário verificar se m, transmitida por p_i, possui o valor do contador esperado como valor da próxima mensagem daquele mesmo processo origem. Caso contrário, haverá uma

Algoritmo 2 Algoritmo para a construção da ordem global.

```
Sender:
 1: Init:
 2:    RBtype := AtomicRB   // Define o algoritmo
 3:    local_seq := 1
 4: upon broadcast(m) do
 5:    broadcast(m, local_seq, RBtype)
 6:    local_seq := local_seq + 1

Sequencer:
 7: Init:
 8:    global_seq := 1   // Contador de mensagens global utilizado para entregar m
 9: upon receive (m, local_seq, RBType) do
10:    broadcast(m, global_seq)
11:    global_seq := global_seq + 1

Receiver:
12: Init:
13:    nextMsg := 1
14:    pendingMsg := ∅
15: upon receive (m, global_seq) do
16:    pendingMsg := pendingMsg ∪ {m}
17:    while (∃ (m′ ∈ pendingMsg ∧ global_seq = nextMsg)) do
18:       deliver(m′)
19:       pendingMsg := pendingMsg \ {m′}
20:       nextMsg := nextMsg + 1
21:    end while
```

pendência na ordem FIFO que precisa ser resolvida. Neste caso, m é inserida no conjunto de pendências ($F\_pendingMsg$). Em outras palavras, se por exemplo, p_i transmitir $m_3^{p_i}$ e $m_4^{p_i}$ (onde $m_3^{p_i}$ é uma mensagem transmitida por p_i e 3 é o valor de local_seq) e o sequenciador receber primeiro m_4, ela armazena e atrasa a transmissão de m_4 até que m_3 seja recebida.

Seguindo o exemplo, quando a respectiva mensagem esperada (m_3) é recebida pelo algoritmo FIFO, ela e as mensagens que estão pendentes (m_4) já podem ser encaminhadas pelo sequenciador. A ordem de encaminhamento das mensagens pendentes respeita a ordem do contador local inseridas em $F\_pendingMsg$. Então, o sequenciador adiciona o valor do contador global à mensagem e transmite $m^s_{m'}$ e $m^s_{m''}$ por difusão confiável aos destinatários, onde $m' < m''$, no exemplo m'=m_3 e m''=m_4. Por fim, os receptores entregam as mensagens de acordo com o Algoritmo 2 (*Receiver*), isto é, enquanto houver mensagens pendentes (mensagens armazenadas em $pendingMsg$) e com o valor do contador global igual ao respectivo valor esperado ($nextMsg$), então as mensagens são entregues nesta ordem para a aplicação.

Observe que neste exemplo se um outro processo p_j transmitir $m_1^{p_j}$ (primeira mensagem de p_j), o *Sequencer*, ao receber esta mensagem, a processa e retransmite imediatamente, pois $m_1^{p_j}$ é independente das mensagens transmitidas por p_i quando considerada a ordem FIFO.

Na estratégia apresentada, como toda mensagem originada de uma difusão ordenada passa pelo sequenciador, a ordem causal é garantida ao garantir a ordem FIFO.

6.3.3 Arquitetura e Implementação do *AnyBone*

A arquitetura do *AnyBone* é composta por componentes que estão operando dentro da própria rede SDN. Para disponibilizar as primitivas de comunicação às aplicações, é oferecida uma API na biblioteca denominada de *RBCast*. A arquitetura pode ser visualizada na Figura 6.2.

Algoritmo 3 Algoritmo para a construção da ordem atômica e FIFO.

```
Sender:
 1: Init:
 2:     RBtype := AtomicFIFO
 3:     local_seq := 1
 4: upon broadcast(m) do
 5:     broadcast(m, local_seq, RBtype)
 6:     local_seq := local_seq + 1

Sequencer:
 7: Init:
 8:     nextMsg := 1
 9:     F_pendingMsg := ∅   // Conjunto de mensagens pendentes
10: upon receive (m, local_seq, RBType) do
11:     if (RBtype = AtomicFIFO) then
12:         if (local_seq = nextMsg) then
13:             broadcast(m, local_seq)
14:             nextMsg := nextMsg + 1
15:             while (∃ (m' ∈ F_pendingMsg ∧ local_seq = nextMsg)) do
16:                 broadcast(m', local_seq)
17:                 nextMsg := nextMsg + 1
18:                 F_pendingMsg := F_pendingMsg \ {m'}
19:             end while
20:         else
21:             F_pendingMsg := F_pendingMsg ∪ {m'}
22:         end if
23:     end if
```

Observe que a *RBCast* está, estrategicamente, nas pontas da comunicação, ou seja, junto aos processos emissores e receptores. A forma como os *hosts* se comunicam é descrita a seguir.

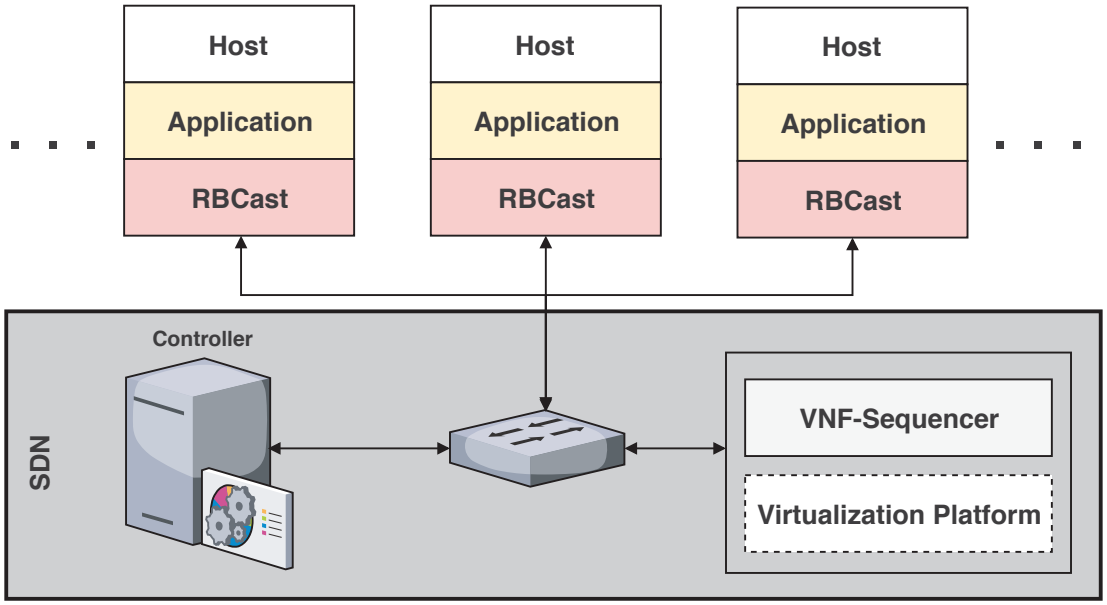


Figura 6.2: Arquitetura do *AnyBone*.

Uma vez que algum *host* deseja realizar uma difusão na rede, é primeiro escolhido o algoritmo de difusão e as mensagens são encaminhadas a um *switch* da rede SDN. Do funcionamento padrão do protocolo *OpenFlow* [McKeown et al. 2008a], toda mensagem que

não possui uma entrada na tabela de fluxos é encaminhada ao controlador. Se o algoritmo escolhido for a difusão confiável, as mensagens não precisam ser enviadas ao sequenciador e são entregues de acordo com o algoritmo descrito em [Chandra and Toueg 1996]. Caso contrário, o controlador SDN é configurado para que as regras instaladas nos *switches* encaminhem o fluxo de pacotes para a própria *VNF-Sequencer*, sempre que a difusão for ordenada. Dessa forma, a *VNF-Sequencer* recebe todo o fluxo de mensagens gerado a partir de uma difusão. Ao receber o fluxo de pacotes, a *VNF-Sequencer* executa o algoritmo especificado pelo processo emissor e envia as mensagens aos processos receptores de acordo com os algoritmos especificados na Seção 6.3.2.

Como relatado por [Défago et al. 2004], em geral o sequenciador é um processo único no sistema. Sendo assim, o *AnyBone* utiliza a premissa de que o sequenciador nunca falha. Além disso, considera-se um canal confiável, *i.e.*, que não cria, não perde e não altera nenhuma mensagem transmitida.

O sequenciador é implementado como uma função virtualizada de rede denominada de *VNF-Sequencer*. A *VNF-Sequencer* explora as funcionalidades da rede SDN, isto é, como há um controlador SDN responsável por gerenciar toda a comunicação da rede, ela recebe do controlador todas as comunicações referentes às mensagens encaminhadas através da biblioteca *RBCast*. *RBCast* é uma biblioteca que possui a implementação das primitivas para a difusão confiável, bem como uma interface para a comunicação com as aplicações. Quando um processo deseja transmitir uma mensagem *m* por difusão confiável e atômica, a ordem total é construída através de dois passos, denominados de *Process-to-Sequencer* (primeiro passo) e *Sequencer-to-Process* (segundo passo), ambas detalhadas a seguir.

Process-to-Sequencer: o emissor de uma mensagem *m* define qual algoritmo será utilizado na transmissão da mensagem. Para isso ele acessa a primitiva de comunicação disponível pela interface *RBCast* e realiza uma difusão dentro da rede SDN. Este processo é similar ao primeiro passo do método BB apresentado na Figura 6.1(b). Do comportamento padrão do protocolo *OpenFlow*, as primeiras mensagens de cada fluxo são encaminhadas para o controlador, para a criação das respectivas regras no *switch*. Neste caso, o controlador é configurado para encaminhar todas as mensagens oriundas da *RBCast* para a *VNF-Sequencer*. Essa estratégia é utilizada para tornar transparente o local de execução da *VNF-Sequencer* (*i.e.*, o emissor não necessita do endereço IP do sequenciador). Alternativamente, pode ser utilizada uma comunicação *unicast* que faz a conexão entre o processo emissor com o sequenciador, de maneira explícita. Por fim, antes da mensagem ser transmitida, *m* recebe um valor de sequência local atribuído pelo próprio emissor.

Sequencer-to-Process: quando a mensagem *m* é recebida pela *VNF-Sequencer*, o sequenciador seleciona o algoritmo para difusão definido no passo *Process-to-Sequencer*, executando todos os procedimentos necessários. Quando *m* é retransmitida pela *VNF-Sequencer* aos processos receptores, a mensagem recebe um valor global de sequência atribuído pelo próprio sequenciador. Por fim, os processos receptores entregam *m* de acordo com o valor de sequência atribuído pela função de rede.

A próxima seção descreve os resultados experimentais executados para a avaliação do funcionamento do *AnyBone* em diversos cenários de execução.

6.4 Avaliação Experimental

Nesta seção são executados experimentos a fim de avaliar o desempenho da *VNF-Sequencer* em uma rede SDN. Além disso, é analisada a viabilidade de mover os serviços de difusão para dentro da rede, de maneira a garantir todas as propriedades de maneira eficiente.

O sistema foi executado em uma máquina física com processador Intel Core i5-7200U@2.50GHz com 4 núcleos, 8 GB de memória RAM e sistema operacional Ubuntu 16.04. Para a geração de mensagens foi desenvolvido um *script* que realiza difusão de mensagens de maneira contínua para as demais aplicações do sistema. As aplicações, a *VNF-Sequencer* e o controlador SDN são executados em *containers* da plataforma Docker. O controlador SDN utilizado é o Ryu. São apresentados experimentos para avaliar o custo da *VNF-Sequencer* em termos de vazão e latência.

6.4.1 Avaliação da Latência da *VNF-Sequencer*

Na Seção 6.3 foi mostrado que o objetivo da *VNF-Sequencer* é garantir a entrega atômica das mensagens em uma rede SDN. A utilização de um sequenciador é, na verdade, a maneira mais eficiente de garantir a ordem total das mensagens. Por outro lado, uma vez que o sequenciador é um único processo que centraliza todas as comunicações, existe um custo computacional decorrente da necessidade de processar todas as mensagens originadas das diversas difusões. Neste sentido, o primeiro experimento tem como objetivo medir o impacto na latência ao realizar uma difusão com sequenciador (difusão atômica) e sem sequenciador (difusão confiável).

No experimento cujos resultados são apresentados na Figura 6.3, a topologia consiste de um *switch*, um controlador SDN e um número variado de processos que participam da difusão confiável. Um dos processos participantes do experimento implementa uma aplicação cliente responsável por inicializar o envio das mensagens que são, inicialmente, repassadas ao *RBCast* local. Quando o *RBCast* dos processos receptores recebe a mensagem, a mensagem é entregue para a aplicação. Neste experimento medimos a latência para a entrega das mensagens, que vai do instante de tempo em que uma mensagem foi transmitida por difusão confiável até o instante em que o último processo tenha entregado a mensagem para a aplicação. Para cada experimento são executadas 1000 difusões com mensagens de 1 KB. Os dados apresentados são valores médios de 10 amostras, utilizando um intervalo de confiança de 95%.

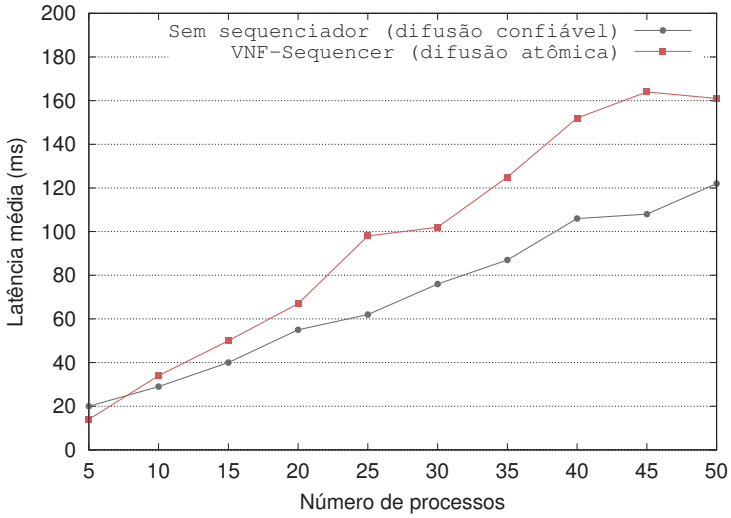


Figura 6.3: Comparação da latência para a entrega das mensagens.

No gráfico da Figura 6.3, o eixo *Y* apresenta o valor da latência em milissegundos para a entrega das mensagens para as aplicações, ao passo que o eixo *X* descreve o número de processos participantes da difusão confiável. A comunicação ocorre com sequenciador (curva *VNF-Sequencer*) e sem sequenciador (curva *Sem sequenciador*). Como previsto, é possível notar

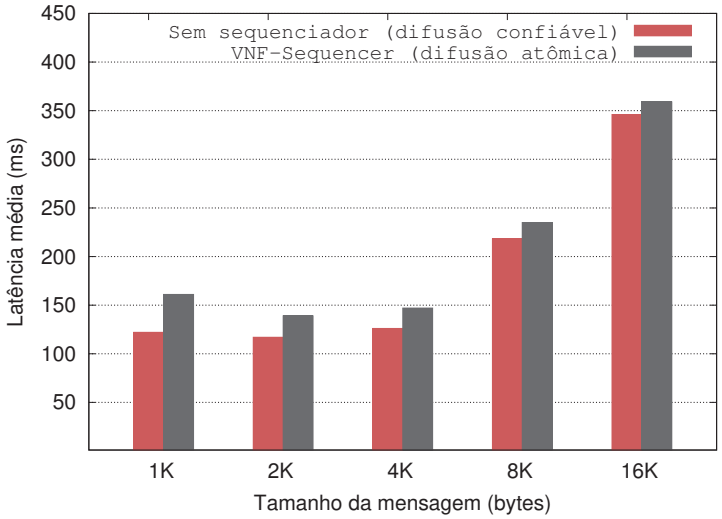


Figura 6.4: Comparação da latência para diferentes tamanhos de mensagens.

que a latência é maior com o uso da *VNF-Sequencer*. Entretanto, o objetivo é verificar quais são os valores deste custo, em termos de tempo. Neste caso, observa-se que os valores da latência ao utilizar um sequenciador é, em média, aproximadamente 32.1% maior. É possível notar também que a latência cresce para ambos os casos conforme o número de processos aumenta. Até 20 processos, o aumento na latência é aproximadamente 20%. Em outras palavras, em um sistema com até 20 processos, o custo para garantir a ordem total das mensagens em todos os processos é relativamente pequeno com o uso de um sequenciador. Para mais do que 20 processos, o custo na latência aumenta de 34% para até 56%.

Foi avaliado também como o tamanho da mensagem influencia no desempenho das difusões. Neste experimento, é analisada a latência para 50 processos, onde um processo cliente envia mensagens com tamanho de 1 KB até 16 KB. A Figura 6.4 mostra a média obtida a partir de 10 amostras, onde cada amostra consiste de 1000 difusões.

Considerando estes resultados, a latência média aumenta em 15.7% com o uso da *VNF-Sequencer*. É importante notar que o aumento na latência diminui conforme o tamanho das mensagens aumenta. Por exemplo, para mensagens de 1 KB, o aumento na latência é aproximadamente 31.7% maior com o uso do sequenciador. Por outro lado, para mensagens de 16 KB, a diferença cai para apenas 3.8%. Em outras palavras, ao aumentar o tamanho da mensagem de 1 KB para 16 KB a latência aumenta em 224 ms sem o uso do sequenciador, ao passo que com sequenciador o aumento é de apenas 198 ms (*i.e.*, 11.6% menor). Portanto, é possível concluir que conforme o tamanho da mensagem aumenta o impacto causado na latência pelo uso da *VNF-Sequencer* é menor.

Diante das análises comparativas, é importante notar que apesar de existir um custo significativo, existe também uma vantagem: a *VNF-Sequencer* implementa a ordem total utilizada na difusão atômica. Por outro lado, a difusão confiável (sem o sequenciador) não garante nenhuma ordem na entrega das mensagens para as aplicações. Portanto, conclui-se que os benefícios proporcionados pelo uso do sequenciador compensam a latência obtida nesta abordagem.

6.4.2 Avaliação da Vazão da *VNF-Sequencer*

O próximo experimento tem por objetivo avaliar a vazão da *VNF-Sequencer*, variando o número de processos participantes da difusão. No experimento, cada execução tem duração de

três minutos e são apresentados dados médios de três execuções. Ao passo em que o número de processos aumenta, o experimento é executado novamente. Dessa forma é calculado, para um intervalo de tempo, quantas mensagens foram processadas no sequenciador e repassadas para os destinatários. Foi avaliada a vazão considerando de 5 até 50 processos, onde um deles é o processo cliente que gera as mensagens. A difusão neste experimento é atômica.

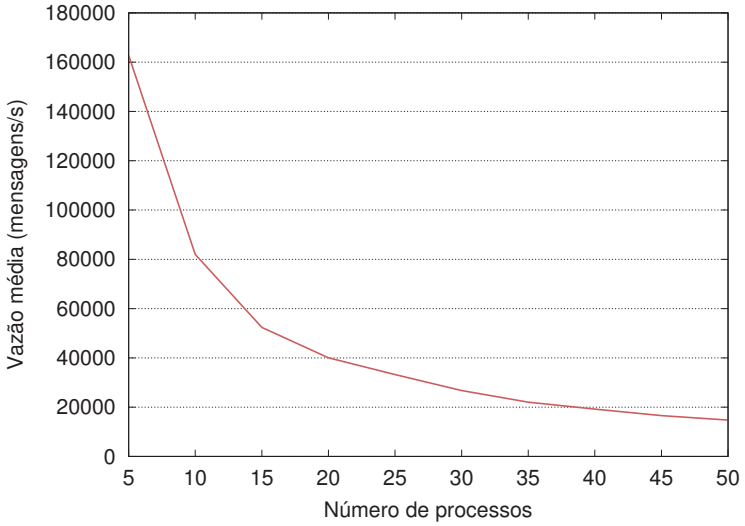


Figura 6.5: Vazão da *VNF-Sequencer* para a difusão atômica.

Na Figura 6.5 é possível observar a variação na vazão conforme o número de processos participantes da difusão aumenta. Uma vez que o processo emissor envia mensagens a uma taxa máxima, a vazão da *VNF-Sequencer* é determinada pelo tempo necessário para enviar as mensagens para todos os demais processos. Destaca-se que a latência da difusão aumenta quando aumenta o número de processos. Como consequência, a vazão diminui conforme o número de processos participantes aumenta. Em especial, para até 50 processos, a *VNF-Sequencer* não mostrou ser um gargalo, alcançando uma vazão de aproximadamente 15000 mensagens por segundo.

Os resultados experimentais demonstrados nesta seção permitem concluir que é possível mover os serviços de difusão para a rede de maneira eficiente e que garanta a entrega confiável e ordenada de mensagens para as aplicações distribuídas.

6.5 Conclusões Parciais

Neste capítulo foi proposto o *AnyBone*: um *backbone* virtual baseado em NFV que oferece as primitivas de difusão confiável para garantir a entrega confiável e ordenada das mensagens transmitidas na rede. Para realizar tal função, são utilizados dois componentes principais: *VNF-Sequencer* que é um sequenciador que gerencia as transmissões e entrega as mensagens ordenadas aos processos; e *RBCast* que oferece uma API para as aplicações trocarem mensagens utilizando as primitivas de difusão confiável. A principal contribuição é transferir para a rede a implementação e disponibilização dos serviços de difusão confiável e ordenada.

Resultados experimentais mostram que a solução proposta cumpre as expectativas de desempenho. O custo da latência foi apresentado em diferentes cenários, ou seja, variando o número de participantes da difusão e o tamanho das mensagens transmitidas. Por fim, foi medida a vazão aumentando o número de processos participantes na difusão atômica. No

experimento observou-se que para até 50 processos a *VNF-Sequencer* não demonstrou ser um gargalo. Uma alternativa para melhorar o desempenho é distribuir a carga em sequenciadores móveis [Défago et al. 2004] ou, até mesmo, implementar sua lógica dentro de dispositivos dedicados como em *switches* proposto em [Li et al. 2016]. Trabalhos futuros também planeja-se implementar a construção da ordem total das mensagens de forma distribuída utilizando consenso.

7 Conclusão

Este trabalho propôs um conjunto de contribuições no contexto de NFV, abordando tanto a gerência do ciclo de vida das VNFs quanto a implementação de serviços distribuídos na própria rede.

Na primeira contribuição foi proposta uma arquitetura de um VNFM que simplifica o gerenciamento do ciclo de vida das VNFs, de forma completa e a permitir o uso de diferentes plataformas NFV. Para tanto, foi utilizado o submódulo *Event Manager* que manipula as três APIs definidas: *Management*, *Resource* e *Function* API. A *Management* API é exposta ao usuário e oferece as funções alto nível para a gerência das VNFs. As outras duas APIs são utilizadas apenas pelo *Event Manager* com o objetivo de automatizar os procedimentos e permitir o uso de diferentes tecnologias NFV. Um protótipo, denominado de *vCommander*, foi implementado, e resultados experimentais demonstram a efetividade da solução.

A segunda contribuição aborda a sincronização consistente de um plano de controle distribuído em uma rede SDN. Foi então proposta uma VNF, denominada de *VNF-Consensus*, que implementa o algoritmo Paxos para garantir a consistência das operações entre múltiplos controladores SDN. Nesta estratégia, os controladores não precisam executar as tarefas de sincronização, que são computacionalmente custosas. Os resultados experimentais comparam a utilização da *VNF-Consensus* com a execução do consenso nos próprios controladores. Em especial, a *VNF-Consensus* permite a sincronização do plano de controle sem aumentar a carga de trabalho nos controladores, o que gera ganhos significativos tanto em vazão quanto em latência.

Por fim, a última contribuição propõe o *AnyBone*, um *backbone* virtual que implementa uma função virtualizada de rede que oferece múltiplos serviços de difusão confiável de forma a garantir a entrega ordenada de todas as mensagens transmitidas na rede. Em especial, o *AnyBone* oferece a difusão atômica, onde todos os processos do sistema entregam as mensagens na mesma ordem. De forma a construir a ordem total das mensagens, o *AnyBone* utiliza um sequenciador, uma entidade centralizada disponibilizada também como uma VNF, denominada de *VNF-Sequencer*. Destaca-se que esta proposta tem como contribuição original a implementação dentro da rede de serviços normalmente executados na camada de aplicação. Outros exemplos de funções virtualizadas de rede da literatura reportam simplesmente a implementação de *middleboxes*.

Para trabalhos futuros relacionados ao *AnyBone*, planeja-se mudar a forma como a ordem total é construída. A utilização de um sequenciador centralizado representa um único ponto de falha, o que consequentemente pode comprometer toda a comunicação da rede. Neste sentido, a ordem total será construída através de um algoritmo de consenso, de forma a tornar o *AnyBone* uma solução tolerante a falhas. Para tanto, a *VNF-Consensus* será incorporada na arquitetura do *AnyBone*. Destaca-se ainda a possibilidade de utilizar os componentes do módulo NFV-MANO para gerenciar e orquestrar o *AnyBone*. Em especial, aplicando técnicas de alocação dinâmica de recursos (*i.e.*, *autoscaling*) com o objetivo de melhorar o desempenho dos algoritmos de difusão.

Referências

[Avizienis et al. 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

[Berde et al. 2014] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., and Parulkar, G. (2014). ONOS: Towards an Open, Distributed SDN OS. In *3th Workshop on Hot Topics in Software Defined Networking (HotSDN)*.

[Birman and Joseph 1987] Birman, K. P. and Joseph, T. A. (1987). Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76.

[Borran et al. 2012] Borran, F., Hutle, M., Santos, N., and Schiper, A. (2012). Quantitative analysis of consensus algorithms. *IEEE Transactions on Dependable and Secure Computing*, 9(2).

[Bosshart et al. 2014] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

[Cachin et al. 2011] Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition.

[Canini et al. 2015] Canini, M., Kuznetsov, P., Levin, D., and Schmid, S. (2015). A distributed and robust SDN control plane for transactional network updates. In *IEEE Conference on Computer Communications (INFOCOM)*.

[Chandra et al. 1996] Chandra, T. D., Hadzilacos, V., and Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722.

[Chandra and Toueg 1996] Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2).

[Chiosi et al. 2012] Chiosi, M., Clarke, D., Willis, P., Reid, A., Feger, J., Bugenhagen, M., Khan, W., Fargano, M., Cui, C., Deng, H., et al. (2012). Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow World Congress*, pages 22–24.

[Correia et al. 2006] Correia, M., Neves, N. F., and Veríssimo, P. (2006). From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96.

[Cotroneo et al. 2014] Cotroneo, D., De Simone, L., Iannillo, A., Lanzaro, A., Natella, R., Fan, J., and Ping, W. (2014). Network function virtualization: Challenges and directions for reliability assurance. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 37–42. IEEE.

[da Cruz Marcuzzo et al. 2017] da Cruz Marcuzzo, L., Garcia, V. F., Cunha, V., Corujo, D., Barraca, J. P., Aguiar, R. L., Schaeffer-Filho, A. E., Granville, L. Z., and dos Santos, C. R. (2017). Click-on-osv: A platform for running click-based middleboxes. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pages 885–886. IEEE.

[Dang et al. 2015] Dang, H. T., Sciascia, D., Canini, M., Pedone, F., and Soulé, R. (2015). Netpaxos: Consensus at network speed. In *Symposium on Software Defined Networking Research, (SOSR'15/SIGCOMM)*.

[de Camargo and Duarte 2017] de Camargo, E. T. and Duarte, E. P. (2017). *Tolerância a Falhas em Sistemas MPI com Grupos Dinâmicos de Processos Recomendados e Registro de Mensagens Distribuído Baseado em Paxos*. PhD thesis, Universidade Federal do Paraná - UFPR, Curitiba - Brasil.

[Défago et al. 2004] Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421.

[Dmitriy Andrushko 2017] Dmitriy Andrushko, G. E. (2017). What is the best nfv orchestration platform? a review of osm, open-o, cord, and cloudify. `https://www.mirantis.com/blog/which-nfv-orchestration-platform-best-review-osm-open-o-cord-cloudify/`. Accessed: 2017-12-14.

[Docker 2017] Docker (2017). Docker - build, ship, and run any app, anywhere. `https://www.docker.com/`. Acessado em novembro de 2017.

[Ekwall and Schiper 2007] Ekwall, R. and Schiper, A. (2007). Modeling and validating the performance of atomic broadcast algorithms in high latency networks. In *13th International Euro-Par Conference*.

[ETSI 2017] ETSI (2017). Open source mano. `https://osm.etsi.org/`. Accessed: 2017-11-21.

[ETSI 2016] ETSI (Available at http://www.etsi.org/technologies-clusters/technologies/nfv, Accessed on October 02, 2016). Etsi gs nfv 002: Architectural framework.

[ETSI 2014] ETSI, N. (2014). Network functions virtualisation (nfv); management and orchestration. *NFV-MAN*, 1:v0.

[ETSI 2015] ETSI, N. (2015). Network functions virtualization (nfv) infrastructure overview. *NFV-INF*, 1:V1.

[Eugster et al. 2004] Eugster, P. T., Guerraoui, R., and Kouznetsov, P. (2004). D-reliable broadcast: A probabilistic measure of broadcast reliability. In *24th International Conference on Distributed Computing Systems (ICDCS)*.

[Felber et al. 1999] Felber, P., Défago, X., Guerraoui, R., and Oser, P. (1999). Failure detectors as first class objects. In *Distributed Objects and Applications, 1999. Proceedings of the International Symposium on*, pages 132–141. IEEE.

[Fischer et al. 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2).

[Franco et al. 2016] Franco, M. F., d. Santos, R. L., Schaeffer-Filho, A., and Granville, L. Z. (2016). Vision – interactive and selective visualization for management of nfv-enabled networks. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 274–281.

[Hadzilacos and Toueg 1994] Hadzilacos, V. and Toueg, S. (1994). A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University.

[Halpern and Pignataro 2015] Halpern, J. and Pignataro, C. (2015). Service function chaining (sfc) architecture.

[Han et al. 2015] Han, B., Gopalakrishnan, V., Ji, L., and Lee, S. (2015). Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97.

[Ho et al. 2016] Ho, C. C., Wang, K., and Hsu, Y. H. (2016). A fast consensus algorithm for multiple controllers in software-defined networks. In *18th International Conference on Advanced Communication Technology (ICACT)*.

[Hunt et al. 2010a] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010a). ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Conference on USENIX Annual Technical Conference (USENIXATC)*.

[Hunt et al. 2010b] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010b). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9. Boston, MA, USA.

[Hwang et al. 2015] Hwang, J., Ramakrishnan, K. K., and Wood, T. (2015). Netvm: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47.

[Kaashoek and Tanenbaum 1991] Kaashoek, M. F. and Tanenbaum, A. S. (1991). Group communication in the amoeba distributed operating system. In *Distributed Computing Systems, 1991., 11th International Conference on*, pages 222–230. IEEE.

[Koponen et al. 2010] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. (2010). Onix: A Distributed Control Platform for Large-scale Production Networks. In *9th Conference on Operating Systems Design and Implementation (OSDI)*.

[Kourtis et al. 2015] Kourtis, M.-A., Xilouris, G., Riccobene, V., McGrath, M. J., Petralia, G., Koumaras, H., Gardikis, G., and Liberal, F. (2015). Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 74–78. IEEE.

[Kshemkalyani and Singhal 2011] Kshemkalyani, A. D. and Singhal, M. (2011). *Distributed computing: principles, algorithms, and systems*. Cambridge University Press.

[Lamport 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

[Lamport 1998] Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2).

[Lamport 2001] Lamport, L. (2001). Paxos Made Simple. *SIGACT News*, 32(4):51–58.

[Lamport 2006] Lamport, L. (2006). Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125.

[Li et al. 2016] Li, J., Michael, E., Sharma, N. K., Szekeres, A., and Ports, D. R. K. (2016). Just say no to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[Li and Crandall 2017] Li, S. and Crandall, J. (2017). TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0. OASIS committee specification draft, Open Standards for the Information Society. `http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html`.

[Martins et al. 2014] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. (2014). Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association.

[McKeown et al. 2008a] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008a). Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.

[McKeown et al. 2008b] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008b). Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.

[Mechtri et al. 2017] Mechtri, M., Ghribi, C., Soualah, O., and Zeghlache, D. (2017). Nfv orchestration framework addressing sfc challenges. *IEEE Communications Magazine*, 55(6):16–23.

[Mijumbi et al. 2016] Mijumbi, R., Serrat, J., Gorricho, J.-L., Bouten, N., De Turck, F., and Boutaba, R. (2016). Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262.

[Mullender et al. 1993] Mullender, S. et al. (1993). *Distributed systems*, volume 12. acm press United States of America.

[ONAP 2017] ONAP (2017). Onap. `https://www.onap.org/`. Accessed: 2017-11-21.

[Ongaro and Ousterhout 2014] Ongaro, D. and Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm. In *USENIX Conference on USENIX Annual Technical Conference (USENIXATC)*.

[OpenBaton 2017] OpenBaton (2017). Openbaton. https://openbaton.github.io/. Accessed: 2017-11-21.

[OpenStack 2017] OpenStack (2017). Openstack. https://www.openstack.org/. Accessed: 2017-11-24.

[OpenVIM 2017] OpenVIM (2017). Openvim. https://www.sdxcentral.com/projects/openvim/. Accessed: 2017-11-24.

[Pedone and Schiper 2003] Pedone, F. and Schiper, A. (2003). Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science (Elsevier)*, 291(1):79–101.

[Rao et al. 2011] Rao, J., Shekita, E. J., and Tata, S. (2011). Using paxos to build a scalable, consistent, and highly available datastore. *The Proceedings of the VLDB Endowment*, 4(4):243–254.

[Reed and Junqueira 2008] Reed, B. and Junqueira, F. P. (2008). A simple totally ordered broadcast protocol. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 2. ACM.

[Rift.io 2017] Rift.io (2017). Rift.io. https://riftio.com/. Accessed: 2017-11-21.

[Sahoo et al. 2010] Sahoo, J., Mohapatra, S., and Lath, R. (2010). Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. In *2010 Second International Conference on Computer and Network Technology*, pages 222–226. IEEE.

[Sasaki et al. 2016] Sasaki, T., Pappas, C., Lee, T., Hoefler, T., and Perrig, A. (2016). Sdnsec: Forwarding accountability for the SDN data plane. In *25th International Conference on Computer Communication and Networks ICCCN*.

[Schiff et al. 2016] Schiff, L., Schmid, S., and Kuznetsov, P. (2016). In-Band Synchronization for Distributed SDN Control Planes. *SIGCOMM Comput. Commun. Rev.*, 46(1).

[Sekar et al. 2012] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. (2012). Design and implementation of a consolidated middlebox architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336.

[Shen et al. 2015] Shen, W., Yoshida, M., Minato, K., and Imajuku, W. (2015). vconductor: An enabler for achieving virtual network integration as a service. *IEEE Communications Magazine*, 53(2):116–124.

[Sherry et al. 2012] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. (2012). Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012*.

[Tacker 2017] Tacker (2017). Tacker. https://wiki.openstack.org/wiki/Tacker. Accessed: 2017-11-21.

[Tianzhu et al. 2016] Tianzhu, Z., Andrea, B., Samuele De, D., and Paolo, G. (2016). The role of inter-controller traffic for placement of distributed sdn controllers. In *Networking and Internet Architecture.*

[Turchetti and Duarte 2015] Turchetti, R. C. and Duarte, E. P. (2015). Implementation of failure detector based on network function virtualization. *Network, iEEE*, pages 19–25.

[Van Renesse and Altinbuken 2015] Van Renesse, R. and Altinbuken, D. (2015). Paxos made moderately complex. *ACM Computing Surveys*, 47(3):42:1–42:36.

[Xilouris et al. 2014] Xilouris, G., Trouva, E., Lobillo, F., Soares, J. M., Carapinha, J., McGrath, M. J., Gardikis, G., Paglierani, P., Pallis, E., Zuccaro, L., Rebahi, Y., and Kourtis, A. (2014). T-nova: A marketplace for virtualized network functions. In *2014 European Conference on Networks and Communications (EuCNC)*, pages 1–5.

[Zhou et al. 2014] Zhou, B., Wu, C., Hong, X., and Jiang, M. (2014). Programming network via distributed control in software-defined networks. In *2014 IEEE International Conference on Communications (ICC)*.

Apêndice A: Publicações

A.1 Trabalhos Publicados no Âmbito da Dissertação

1. **Venâncio, G.**, Garcia, V., Marcuzzo, L., Tavares, T., Franco, M., Bondan, L., Schaeffer-Filho, A., Santos, C., Granville, L. e Duarte, E. P. (2018). Simplificando o Gerenciamento do Ciclo de Vida de Funções Virtualizadas de Rede. Em XXIII Workshop de Gerência e Operação de Redes e Serviços (WGRS). XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'18). (Prêmio de melhor artigo)

2. Turchetti, R. C., **Venâncio, G.** e Duarte, E. P. (2018). Utilizando NFV para Implementar a Difusão Confiável e Ordenada de Mensagens na Rede. Em XIX Workshop de Testes e Tolerância a Falhas (WTF). XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'18).

3. **Venâncio, G.**, Turchetti, R. C. e Duarte, E. P. (2018). On-The-Fly NFV: Em Busca de uma Alternativa Simples para a Instanciação de Funções Virtualizadas de Rede. Em IX Workshop De Pesquisa Experimental Da Internet Do Futuro (WPEIF). XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'18).

4. **Venâncio, G.**, Turchetti, R. C., de Camargo, E. T. e Duarte, E. P. (2017). Uma Função Virtualizada de Rede para a Sincronização Consistente do Plano de Controle em Redes SDN. XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'17).

A.2 Trabalhos Publicados no Âmbito do Grupo de Pesquisa

1. Bondan, L., Franco, M., **Venâncio, G.**, Marcuzzo, L., Schneider, C. A. D. S., Duarte, E. P., Schaeffer-Filho, A., Santos, C., Granville, L. (2018). FENDE: Marketplace and Federated Ecosystem for the Distribution and Execution of VNFs. In ACM Special Interest Group on Data Communication Posters and Demos (SIGCOMM'18).

2. Huff, A., **Venâncio, G.**, Marcuzzo, L., Garcia, V., Santos, C. e Duarte, E. P. (2018). A Holistic Approach to Define Service Chains Using Click-on-OSv on Different NFV Platforms. In IEEE Global Communications Conference (GLOBECOM'18).

3. Huff, A., **Venâncio, G.**, Marcuzzo, L., Garcia, V., Santos, C. e Duarte, E. P. (2018). Uma Abordagem Holística para a Definição de Service Chains Utilizando Click-on-OSv sobre Diferentes Plataformas NFV. Em XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'18).

4. Tavares, T. N., Marcuzzo, L. C., Garcia, V. F., **Venâncio, G.**, Franco, M. F., Bondan, L., Turck, F., Granville, L. Z., Duarte, E. P., Santos, C. R. P. e Schaeffer-Filho, A. E. NIEP: an NFV Infrastructure Emulation Platform. In IEEE International Conference on Communications (AINA'2018).